
Computer Science

Scalability in Production System Programs

Anurag Acharya

November, 1994

CMU-CS-94-211

19941228 128

**Carnegie
Mellon**

Keywords: production system programs, scalability, scalable parallelism, scalable match algorithms, collection-oriented match

Scalability in Production System Programs

Anurag Acharya

November, 1994

CMU-CS-94-211

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Thesis Committee

Peter Lee, Chair

Guy Blelloch

Jim Larus

Paul Rosenbloom

Milind Tambe

Accepted For	
MHS	CR 24
B7C	1, 3
L	1, 2
J	
S	
D, 1	
D, 2	
D, 3	
D, 4	
A-1	

Keywords: production system programs, scalability, scalable parallelism, scalable match algorithms, collection-oriented match



School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

SCALABILITY IN PRODUCTION SYSTEM PROGRAMS

ANURAG ACHARYA

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

P. Lee 11/17/94
THESIS COMMITTEE CHAIR DATE

J. Morris 11/17/94
DEPARTMENT HEAD DATE

APPROVED:

D. R. J. 11/15/94
DEAN DATE

Abstract

Production system programs have been notorious for their inability to handle large data sets. The primary cause of their poor scalability is the combinatorial explosion in the number of possible matches which arises from the need to match conjunctive conditions where each conjunct can match the whole data set. This dissertation investigates two approaches for handling large data sets in production system programs - scalable parallelism and scalable match algorithms.

The primary limitation on parallelism in production system programs is the data-dependent nature of the computation combined with a lack of information about the run-time contents of the tuple-space. This dissertation argues that effective parallelization of production system programs requires information about the run-time contents of the tuple-space. Results from a simulation study show that simple extensions to existing production system languages can provide sufficient information to parallelize a wide variety of programs. These results also show that, in general, there is no program-independent bound on the speedup that can be achieved by parallel production system programs and that speedups in such programs can scale with data set size.

This dissertation describes a new approach to matching, referred to as *collection-oriented match*, which attempts to mitigate the combinatorial explosion in the number of possible matches by grouping tuples that cannot be distinguished by the conditions in the program. The rate of growth in number of possible matches in a collection-oriented match algorithm depends on the extent to which the tuples matching individual conditions can be grouped together. From a database point of view, collection-oriented match uses lazy instead of eager joins. Since collection-oriented match imposes no restriction on the expressiveness of the productions or the contents of the tuple-space, it does not reduce the worst-case space and time complexity of the production match problem. But, as the empirical results presented in this dissertation show, it can dramatically increase the scalability of a wide variety of production system programs.

Acknowledgements

Most of the summer of 1971, I spent in a room that went tip-tap all day. That was a magic summer. By the time the monsoons came splashing down, my uncle had put together enough sheets of paper to earn the elite two-letter prefix. And by diligent monitoring of the waste paper basket, I had a hefty volume of my own. Complete with color figures. Enthusiastically, I showed it to all visitors. And took all the ribbing seriously. It is the end of another summer and I have another volume. It has weight, it has figures, it spellchecks, it will get me my own little prefix. Some time soon, I will put on a clean shirt and walk into the concrete cavern of 4623. But my heart will be in that little room twenty-three years and ten thousand miles away. Peering over the shoulder of its occupant who started me on this journey. Dr. Harnarayan Acharya, thank you very much.

When I came to CMU, it was to "do hardware", whatever that meant. But then I met this tall, stooping, mostly bald man who glared at me through large glasses. I guess I must have liked being glared at because within a few days, he was my advisor and I was off into the land of artificial intelligence trying to figure out how people were able to use hand calculators. I never did figure that out. But for the next five years, I rode a roller-coaster of research. And he rode with me. Through all the ups and downs. As he said in his book, *Unified Theories of Cognition*, for him, theories and graduate students were alike – once admitted he tried hard to help them succeed, it being, much better for them and for the world if they could become long-term contributors to society. For those who know what I mean, I could do with an Allen Newell handshake today.

For a short while in the Spring of 1993, the good old CMU computer science department decided to stop mollycoddling its graduate students for once and exposed them to the murky games that academics play. Finding funding, keeping it going, walking the line between academic soberness and mercantile advertisement. For a few lectures, prospects seemed rather bleak. It seemed like a Faustian bargain was inevitable in these days of big science. Not that this was entirely a surprise but it was rather disappointing to be told so in a matter-of-fact sort of a way. But I despaired too soon. Next week brought better tidings. That it was possible, even these days, to save one's soul and yet have an scientific impact. It was vitally important, the speaker told us, that we remember who we were. Scientists first and other things later. It washed away the bad taste in my mouth. I am proud that the speaker was my advisor and friend, Peter Lee.

A lazy summer afternoon, the cool shady wall of Central Park, coin weighing problems. How many weighings do you need for thirteen coins? How many for nineteen? What if you could lie a finite number of times? Do knights and knaves enter in this somewhere? And if knights and knaves are arcund, can ladies and tigers be far away? It has been a wonderful year and half. You know who you are, pal.

I have spent seven years, two months and a handful days in grad school. For six of those, I shared my apartment with the same person. They tell me it is impossible to share a house with

anyone that long. I guess, they don't know Puneet. We have grown together (though each in our own way). If you do happen to meet him somewhere, remember he is the best Indian cook you will meet in your life.

When I was applying to grad schools, CMU has a special attraction. Its brochure talked about "being reasonable", being flexible and being supportive and being exciting. It has been all four. Even though everything else around has changed so much. I have now spent about one-fourth of my life here. I have drunk deep at the spring. I am glad. It is a wonderful place.

There are so many people that I met here. If I talk about all of them, this acknowledgement will probably end up being longer than the thesis itself (long as it is). But, it is they who helped me keep my sanity through all these years. Thank you!

Contents

1	Introduction	1
1.1	Preview of Results	3
1.2	Map of the dissertation	5
2	Background	7
2.1	Production Systems	7
2.2	OPSS	8
2.3	Match Algorithms	11
2.4	Rete	13
2.5	Parallel Rete	17
3	Design and Implementation of PPL	20
3.1	Need for Parallel Production Languages	20
3.1.1	Need for barrier synchronization	21
3.1.2	Frequency of barrier synchronization	23
3.1.3	Desiderata for a parallel language	29
3.2	Design Space of Parallel Production Languages	29
3.2.1	Specification of ordering relations	31
3.2.2	Which instantiations are comparable	32
3.3	PPL	34
3.3.1	Parallel constructs	34
3.3.2	Expressiveness	36
3.3.3	Comparison with other parallel constructs	37

CONTENTS

v

3.4	Implementation of PPL	39
3.4.1	Overall organization	39
3.4.2	The <i>pplc</i> compiler	40
3.4.3	The <i>pplc</i> run-time library	45
3.4.4	Parallel state-maintenance algorithms	46
3.4.5	Minimizing resource contention	47
4	Parallelism Experiments	50
4.1	Benchmark suite	50
4.1.1	Circuit simulator (<i>circuit</i>)	51
4.1.2	Game of Life (<i>life</i>)	52
4.1.3	Waltz labeling (<i>waltz</i>)	54
4.1.4	Simulation of a hotel (<i>hotel</i>)	55
4.1.5	Interpretation of aerial images (<i>spam</i>)	56
4.2	Design of the experiments	59
4.3	Simulator	62
4.3.1	Structure and operation of the simulator	63
4.3.2	Limitations of the simulator	69
4.3.3	Validity of the simulator	72
5	Parallelism Experiments: Results, Analysis and Observations	74
5.1	Speedups	74
5.1.1	Analysis	76
5.2	Growth of speedups with data set size	86
5.2.1	Growth of speedups for <i>circuit</i>	87
5.2.2	Growth of speedups for <i>life</i>	89
5.2.3	Growth of speedups for <i>waltz</i>	91
5.2.4	Growth of speedups for <i>hotel</i>	94
5.2.5	Growth of speedups for <i>spam</i>	99
5.3	Conclusions and observations	101
5.3.1	Validation of hypotheses	101

5.3.2	Aggregate updates faster with parallel constructs	102
5.3.3	Recency unsuitable for parallel languages	102
5.3.4	Multiple copies of parallel productions are desirable	106
5.3.5	Guidelines for programming parallel production system languages	108
5.3.6	Collection-oriented semantics essential for scalable parallelism	113
5.3.7	Performance on real machines	116
6	Collection-oriented Match	121
6.1	The key idea	122
6.1.1	Analysis	124
6.2	Collection-oriented match algorithms	125
6.2.1	Right memory nodes	126
6.2.2	Left memory nodes	127
6.2.3	Pnodes	129
6.3	Collection Rete	129
7	Collection-oriented Production Language	139
7.1	Design of the Collection-oriented Production Language	139
7.1.1	Desiderata	139
7.1.2	Tuple space	141
7.1.3	Conditions and Instantiations	141
7.2	Implementation of COPL	145
7.2.1	Data structures	146
7.2.2	Procedures	147
8	Collection-oriented Match Experiments	149
8.1	Benchmarks	149
8.1.1	Creating teams with constraints (make-teams)	150
8.1.2	Clustering image regions (clusters)	153
8.1.3	Airline routing (airline-route)	156
8.2	Design of the experiments	158

8.3	Comparison between PPL and COPL	158
8.3.1	Analysis	158
8.3.2	Critique	164
8.4	Scalability of collection-oriented match	165
8.4.1	Scalability of Collection Rete	167
8.5	Observations	171
8.5.1	Relational match tests are inefficient	171
8.5.2	Negation needed infrequently	173
8.5.3	Condition ordering less important for efficiency	174
8.5.4	Cardinality of variable values depend on condition ordering	175
8.5.5	Interaction with parallelism	176
8.5.6	Less restrictive is better	177
8.5.7	Programming guidelines	177
8.5.8	It is possible to be even more lazy	179
9	Related Work	181
9.1	Parallel match	182
9.1.1	Tree-structured architectures	182
9.1.2	Data-flow architectures	183
9.1.3	Shared memory architectures	185
9.1.4	SIMD architectures	185
9.1.5	Message-passing multicompilers	186
9.1.6	Conclusions	186
9.2	Parallel firings	187
9.3	Parallel Production Languages	191
9.4	Reducing Combinatorics	196
9.4.1	Relaxing the completeness constraint	196
9.4.2	Reduce the power of each condition	197
9.4.3	Relax the correctness constraint	203

10 Conclusions	205
10.1 Primary conclusions	205
10.2 Some general conclusions	209
10.3 Directions for future research	212
A Trace formats used in the parallelism experiments	215
A.1 Static trace	215
A.1.1 Information about individual production-sets	216
A.1.2 Information about individual functions	216
A.1.3 Information about individual nodes	217
A.2 Dynamic trace	217
B Cost model used in parallelism experiments	221
C Configuration file for the simulator	229
D Detailed parallelism results	231
E Code for benchmarks used in parallelism experiments	245
E.1 Sequential version of the game of life	245
E.2 Parallel version of the game of life	251
E.3 Sequential version of the circuit simulator	257
E.4 Parallel version of the circuit simulator	267
E.5 Sequential version of waltz	277
E.6 Parallel version of waltz	280
E.7 Sequential version of hotel	283
E.8 Parallel version of hotel	299
F Code for benchmarks used in COM experiments	320
F.1 Tuple-oriented version of make-teams	320
F.2 Collection-oriented version of make-teams	322
F.3 Tuple-oriented version of clusters	323

CONTENTS

ix

F.4	Collection-oriented version of clusters	326
F.5	Tuple-oriented version of airline-route	328
F.6	Collection-oriented version of airline-route	331

List of Figures

2.1	High-level view of a production system program	8
2.2	Sample OPS5 tuple	9
2.3	Sample OPS5 production	9
2.4	An instantiation for the sample production	10
2.5	Productions to implement a simulator for 2-input xor-gates	14
2.6	Rete network for the XOR gate productions	15
2.7	Instantiation generated for the xor-gate simulator	17
3.1	Simple production that interferes with itself	22
3.2	Execution of three match-select-act cycles in parallel	25
3.3	Dependency-graph for xor-gate simulator	26
3.4	OPS5 example of indirect dependency	28
3.5	Partially ordered conflict set	31
3.6	Effect of modifying an unstructured partial order	33
3.7	PPL version of the xor-gate simulator	35
3.8	Stripped down code for pipeline parallelism	37
3.9	Fibonacci using a task-pool approach	38
3.10	xor-gate simulator in NESL	39
3.11	Example of the <i>canonical conditions</i> transformation	42
3.12	Example of constraint propagation	42
3.13	Example of a production with inconsistent conditions	43
3.14	Example of type inference	44
3.15	Example of hashed a network	45

3.16 Example of a two-level heap-based conflict set	47
4.1 Linear feedback shift register of size three	52
4.2 The basic pattern for the <i>life</i> data set	54
4.3 The basic block for the <i>waltz</i> data set	55
4.4 Sample production from first phase of SPAM	58
4.5 Sample production from second phase of SPAM	59
4.6 C struct declaration for a trace record	64
4.7 Effect of processing order on token cost	66
4.8 Disassembled multiprocessor code for <i>ppl.add_to_tuple_space()</i>	68
4.9 Disassembled uniprocessor code for <i>ppl.add_to_tuple_space()</i>	69
4.10 C code for <i>ppl.add_to_tuple_space()</i>	70
4.11 Abbreviated simulator output for one of the experiments	71
5.1 Speedups for the comparative suite.	76
5.2 Real and nominal speedups for the comparative suite	78
5.3 Non-parallelizable doubly nested print loop from <i>life</i>	79
5.4 Non-parallelizable loop from third phase of <i>spam</i>	81
5.5 Productions with guard conditions for sequencing	83
5.6 Rete network for the productions with guard conditions	84
5.7 Constraint application production from <i>waltz</i>	84
5.8 Processor utilization for the comparative instance of <i>waltz</i>	85
5.9 Conversion of a linear Rete network to a constrained bilinear network	86
5.10 Speedup curves for <i>circuit</i>	88
5.11 Saturation speedups for <i>circuit</i>	88
5.12 Parallelization overheads for <i>circuit</i>	89
5.13 Nominal speedups for <i>circuit</i>	90
5.14 Estimated maximum nominal speedups for <i>circuit</i>	90
5.15 Speedup curves for <i>life</i>	92
5.16 Saturation speedups for <i>life</i>	92
5.17 Speedup curves for <i>waltz</i>	93

5.18 Saturation speedups for <i>waltz</i>	94
5.19 Speedup curves for <i>hotel</i>	95
5.20 Saturation speedups for <i>hotel</i>	96
5.21 Number of instantiations fired per cycle for <i>hotel</i>	96
5.22 Example of an accumulation loop	97
5.23 Growth of average tasks/cycle in <i>hotel</i>	98
5.24 Growth of average task size in <i>hotel</i>	99
5.25 Speedup curves for <i>spam</i>	100
5.26 Atomic update of an unbounded aggregate (sequential).	103
5.27 Atomic update of an unbounded aggregate (parallel).	103
5.28 Productions illustrating race conditions due to recency.	105
5.29 Productions illustrating use of additional conditions to ensure responsiveness. .	105
5.30 Production to simulate an and-gate.	106
5.31 Productions to simulate an and-gate.	107
5.32 Elimination of flag fields in parallelizable loops	108
5.33 Use of total order to choose one item from a set.	109
5.34 Use of numerical identifiers to eliminate interfering instantiations.	109
5.35 Sequential example using a single tuple for the entire data structure.	110
5.36 Parallel example using partitioned tuples.	111
5.37 Production system and C code to sum a sequence.	112
5.38 Instantiations for tuple-oriented and collection-oriented semantics	114
5.39 Summing a sequence in a collection-oriented production language.	115
6.1 Example production and tuple-space.	122
6.2 SQL version of the example production.	124
6.3 Example production with a negated condition.	128
7.1 Example to illustrate variable binding	140
7.2 Example of binding collections to variables	142
7.3 Example of instantiation ordering in COPL	142
7.4 Production that generates a cross-product in the tuple-space	143

7.5	Production that does not generate a cross-product in the tuple-space	144
7.6	Cross-product and non-cross-product makes in COPL	144
7.7	Example of <code>modify</code> in COPL	145
8.1	PPL productions for make-teams	151
8.2	COPL productions for make-teams	152
8.3	PPL productions for clusters	154
8.4	COPL productions for clusters	155
8.5	PPL productions for airline-route	157
8.6	COPL productions for airline-route	157
8.7	Execution time for make-teams	159
8.8	Match space for make-teams	159
8.9	Execution time for clusters	160
8.10	Match space for clusters	160
8.11	Execution time for airline-route	161
8.12	Match space for airline-route	161
8.13	Example production and tuple-space	166
8.14	Tuple processing rate for make-teams	168
8.15	Tuple processing rate for clusters	169
8.16	Addition of new flights to existing flight database	170
8.17	Tuple processing rate for airline-route	171
8.18	Finding the minimum element using relational tests	172
8.19	Finding the minimum element using a procedure	172
8.20	Programming idioms using negation	173
8.21	Programming idioms rewritten in COPL	174
8.22	Example illustrating the reordering optimization	175
8.23	Example illustrating difference in cardinality of variable values	176
8.24	Decreasing restrictiveness reduces number of instantiations	178
8.25	Production that finds leaves two hops away from the root	180
8.26	Example network	180

9.1	The Pesa - 1 architecture	184
9.2	Dependency-graph for xor-gate simulator	188
9.3	OPS5 example of indirect dependency	190
9.4	Example of nondeterminism in asynchronous firings.	192
9.5	Example to illustrate differences between synchronous languages	193
9.6	Simple production that interferes with itself	195
9.7	PPL productions for make-teams	197
9.8	Example production and tuple-space for multi-attribute representation	199
9.9	Tuple-space and one of the productions for the unique-attribute representation	200
9.10	Before the application of copy-and-constrain	201
9.11	After the application of copy-and-constrain	201
9.12	Data partitioning example	202
9.13	Case whether data partitioning does not help	203
9.14	Constraint graph for the example production	204
10.1	Production that finds leaves two hops away from the root	212
10.2	Example network	213

List of Tables

4.1	Comparison of uniprocessor PPL and CParaOPSS for fixed data set benchmarks	61
4.2	Comparison of uniprocessor PPL and CParaOPSS for variable data set benchmarks	61
4.3	Comparison of predicted and actual running times	73
5.1	Data sets for the comparative suite	75
5.2	Highest speedups achieved	75
5.3	Parallelization overhead	77
5.4	Distribution and average size of tasks in the comparative suite	77
5.5	Average size of <i>msa</i> cycles	80
5.6	Mean and standard deviation for number of conditions per production	82
5.7	Instantiations generated per cycle for the comparative suite	82
5.8	Size of the <i>spam</i> data sets	100
5.9	Saturation speedups for <i>spam</i>	100
5.10	Number of iterations of <i>spam</i> loops	101
5.11	Results for <i>life</i> on an Omron	117
5.12	Results for <i>circuit</i> on an Omron	117
5.13	Results for <i>hotel</i> on an Omron	117
5.14	Results for <i>hotel</i> for a modified PPL implementation	118
5.15	Results for coarse-grain decomposition of <i>circuit</i>	118
5.16	Comparison of estimated and actual execution times on a uniprocessor	119
8.1	Largest experiments	171
10.1	Comparison of ratio of <i>msa</i> cycles and speedups	211

10.2 Comparison average cycle size of sequential and parallel versions	211
D.1 Language level results for spam	232
D.2 Language level results for life	232
D.3 Language level results for circuit	233
D.4 Language level results for waltz	233
D.5 Language level results for hotel	234
D.6 Task breakdown for spam	235
D.7 Task breakdown for life	236
D.8 Task breakdown for circuit	237
D.9 Task breakdown for waltz	238
D.10 Task breakdown for hotel	239
D.11 Execution time breakdown for spam	240
D.12 Execution time breakdown for life	241
D.13 Execution time breakdown for circuit	242
D.14 Execution time breakdown for waltz	243
D.15 Execution time breakdown for hotel	244

Chapter 1

Introduction

The chief characteristic of the production system (rule-based) computational model is the data-dependent nature of its control-flow. A production system program checks its data every so often to determine what to do next. This feature makes the production system computational model attractive for programs in which the sequence of operations is not known *a priori* and needs to be determined dynamically depending on the contents of the data. It is then not surprising that, for a long time, production system programs were almost exclusively used for artificial intelligence programs including cognitive modelling [3, 89, 87, 88, 99], problem-solving systems [57, 98, 125] and expert systems [12, 59, 62, 72, 123]. This flexibility comes with a price tag – production system programs have always been considered slow. For a long time, this limited the applicability of production systems. Over the last two decades, considerable research has been devoted to efficient implementations of production system languages [23, 27, 29, 38, 49, 74, 78, 79, 96, 114]. The combination of better algorithms, efficient compilation techniques and faster hardware platforms has yielded several orders of magnitude speedup. Today, implementations of production system languages are available from several vendors and are used for a variety of expert system programs.

The speedup has also increased the attractiveness of production system programs for other pattern-directed programs. Production systems have been proposed as a single uniform mechanism for diverse tasks in relational database systems – enforcing integrity constraints, monitoring data access and evolution, maintaining derived data, enforcing security schemes, maintain version histories, implement alerters and triggers that initiate actions in the presence of specific data patterns [46]. Integrating production systems and relational database systems is currently a focus of research among database researchers. Several prototypes are under development, e.g., Starburst [127], POSTGRES [108], Ariel [45], RPL [21]. Several commercial relational databases, e.g., Sybase [110], Oracle, Rdb and INGRES [54] provide some, albeit limited, support for rules. Another application area is image-interpretation where production systems are used to coordinate and control image segmentation, segmentation analysis and the construction

of a scene model. An example is the System for Photo-interpretation of Airports using MAPS (SPAM) system [75] developed at Carnegie Mellon. Production systems are also being used in simulation [70, 95, 97] and monitoring large processes [67]. The common characteristic of these applications is that they process orders of magnitude more data than the traditional applications – typical databases contain millions of tuples, large databases even more. Production system implementations, however, have been notorious for their inability to handle large amounts of data. Production system programming texts like *Programming Expert Systems with OPS5* [11] devote several pages to tricks to avoid excruciating slow-downs as the size of the data increases. Production system programs that need to process large amounts of data usually have to be modified to partition the data and process it piecemeal, for example SPAM [47] and Alexsys [107].

This dissertation investigates two approaches to tackle the growth in execution time due to growing data sets - scalable parallelism and scalable match algorithms.

Several research efforts have investigated parallelism in production system programs [2, 14, 38, 42, 48, 51, 56, 61, 84, 86, 90, 105, 128]. These investigations indicated that the amount of parallelism available in production system programs is small. Based on a detailed analysis of a set of six programs of various sizes and organizations, Anoop Gupta [38] concluded that there is an empirical *program-independent* bound of between 20 and 30 fold on the parallelism available in production system programs. He further concluded that fine-grain decomposition is required to achieve significant speedup and that the communication and scheduling overheads of such decompositions limit the achievable speedup to under 20 fold. Results of the other investigations have borne out this conclusion. The primary cause for this program-independent bound is the uniformly high frequency of barrier synchronizations in parallel execution of production system programs. Investigations studying parallel implementations of OPS5 and Soar report that a large fraction of the barrier synchronizations occur after less than 125,000 instructions [42, 113].¹ Since little work is done between successive barrier synchronizations, the number of processors that can be effectively utilized is bounded. Barrier synchronizations are necessary in production system programs to ensure that all changes to the data are completed before trying to determine what to do next. Barrier synchronizations are, thus, the price paid for dynamic determination of control-flow. Contemporary production system languages are geared towards highly dynamic control-flow. They limit the amount of work that can be done between successive branch points. It is this limit that leads to the high rate of barrier synchronizations.

Eliminating the limit on the amount of work between successive barrier synchronizations is necessary for scalable parallelism but not sufficient. It is also necessary to ensure that available parallelism between barrier synchronizations keeps pace with the growth in the amount of work.

This dissertation addresses the problem of scalable parallelism in production system programs

¹250 tasks of between 100 and 500 instructions

at three levels: design of an explicitly parallel language, a fully parallelized implementation and parallel programming idioms.

The primary cause of poor scalability of the match algorithms used in production system implementations is the combinatorial explosion in the number of partial and complete matches [38, 80, 116] which arises from the need to match conjunctive rule conditions. Since every conjunct can potentially match the entire data set, the number of tests and matches, in the worst case, is $O(|D|^n)$ where $|D|$ is the size of the data set and n is the maximum number of conjuncts in the rules.

Research efforts aimed at developing match algorithms with better scaling characteristics have either imposed semantic restrictions on the possible matches and leveraged the restrictions to limit the number of possible matches [79, 116] or they have focused on efficiently managing the large number of matches generated [92, 47, 104, 107, 118]. This dissertation attempts to answer the question whether it is possible to improve the scalability of match algorithms without imposing any semantic restrictions.

1.1 Preview of Results

The primary conclusions of this dissertation are:

1. In general, there is no program-independent bound on the speedup that can be achieved by parallel production system programs. Detailed simulation results presented in this dissertation indicate that speedups up to 115 fold with 200 processors can be achieved and that this is not an upper bound on speedups. Analysis of the results identifies small task size, non-parallelizable loops and large cross-products arising due to the use of sequencing tuples as the primary limitations on speedups. These limitations can be alleviated, if not eliminated, by using collection-oriented match algorithms and collection-oriented languages.
2. Speedups in parallel production systems can scale with data set size. That is, parallelism is a feasible solution for the problem of dealing with large tuple-spaces. Results presented in this dissertation show that in many cases, the speedup achieved with a given configuration grows with the data set size, the rate of growth depending on program characteristics, in particular, the fraction of time spent in non-parallelizable loops, the rate of growth of task size with a growth in data set size and the structure of the dependencies between match tasks.
3. Effective parallelization of production system programs requires information about the run-time contents of the tuple-space. The limited success of automatic parallelization

can be attributed to the data-dependent nature of the computation in production system programs combined with a lack of information about the contents of the tuple-space. In the absence of user specification, there is no way for a production system implementation to obtain this information. Compile-time analyses have access to only the productions and are forced to be overly conservative. Run-time analyses operate within the context of a particular tuple-space but have access only to the instantiations that are present in the conflict set at any given time. To ensure correctness, an implementation based on a run-time analysis would, in general, need to lookahead, possibly to the end of the execution. This could be prohibitively expensive even for modest-sized programs.

4. A fixed ordering procedure combined with program annotations that specify which instantiations are comparable is sufficient for the expression of parallelism in production system programs. Since the annotations are specified at compile-time, they cannot discriminate between different instantiations of a single production. Either all instantiations of a production are incomparable or none of them are. Program-specific ordering procedures that are executed at run-time have access to the instantiations and can be more discriminating. Results presented in this dissertation show that, for a wide variety of programs, the additional power provided by program-specific ordering procedures is not necessary for the expression of parallelism present in the programs.
5. It is sometimes possible to tame the combinatorial explosion in the number of instantiations and partial matches as the data set size grows without restricting either the expressiveness of the productions or the contents of the tuple-space. That is, collection-oriented match is a feasible solution for the problem of dealing with large tuple-spaces. In the best case, the number of instantiations and partial matches can be independent of the data set size; in the worst case, there is no reduction in the number of instantiations and partial matches. Results presented in this dissertation show that for one of the benchmark programs, it is possible to process over 10 million tuples under four minutes on a Decstation 5000/260.
6. The rate of growth in number of instantiations and partial matches in a collection-oriented match algorithm depends on the extent to which the tuples matching individual conditions can be grouped together. If all the tuples matching every condition form a single group, only one instantiation is generated for every production. On the other hand, if every tuple forms its own group, a collection-oriented algorithm reduces to its tuple-oriented analogue. In other words, the number of instantiations for a production depends on how the collections of tuples corresponding to each condition are partitioned, or *fragmented*. Therefore, the rate at which the number of instantiations and tokens grows is governed by the rate at which new partitions are generated.

1.2 Map of the dissertation

Chapter 2 provides the background necessary for the rest of the dissertation. The first section provides a brief introduction to the production system paradigm. The second section describes the syntax and semantics of OPS5. Both the languages designed as a part of this investigation are extensions of OPS5. Furthermore, all the test programs used in this investigation were originally written in OPS5. The third section briefly describes efficient match algorithms and goes into more detail for one of them, Rete. Several schemes have been suggested for parallelizing Rete. The fourth section describes the most successful scheme.

Chapters 3,4 and 5 describe the scalable parallelism investigation. The goal of this investigation was to test three hypotheses. First, that there is no general program-independent bound on the parallelism available in production system programs. Like in other paradigms, the parallelism available in production system programs depends on the parallelism inherent in the program and the way the program has been encoded. Second, that the parallelism available in a production system program can scale with data. That is, parallelism is a possible solution for the problem of dealing with large data sets. Third, that simple extensions to existing production system languages are sufficient for the expression of parallelism in production system programs. Chapter 3 establishes the need for parallel production system languages and describes the design and implementation of one such language. Chapter 4 describes the experiments conducted to evaluate this language and Chapter 5 presents the results, analysis and some observations.

Section 3.1 describes the limitations of automatic parallelization for production system programs and establishes the need for parallel production system languages. Section 3.2 lays out the design space for these languages and evaluates the alternatives. Section 3.3 describes the design of a particular language, Parallel Production Language (PPL). Section 3.4 describes the PPL implementation. Section 4.1 describes the benchmark suite. It describes the programs, the parallelization strategy and the data sets used in the experiments. Section 4.2 describes the structure of the experiments. Section 4.3 describes the simulator used in these experiments to simulate the execution of PPL programs on a multiprocessor. Section 5.1 presents speedups for the full benchmark suite. Section 5.1.1 analyzes the results to identify parallelization overheads, non-parallelizable loops and dependencies between tasks as the major limitations on speedups in parallel production system programs. Section 5.2 shows how the speedup varies with data set size for individual benchmarks. Section 5.3 presents some observations from the experiments, including programming idioms for parallel production system languages and practical advice for parallelizing sequential production system programs. It also argues that collection-oriented match algorithms can be expected to alleviate the limitations on speedups discussed in Section 5.1.1.

Chapters 6,7 and 8 describe the investigation into scalable match algorithms. Chapter 6 identifies the *tuple-oriented* nature of contemporary match algorithms as the primary cause of the combinatorial explosion in the number of instantiations and partial matches. It presents a

new approach to matching, referred to as *collection-oriented match*, which attempts to mitigate the combinatorial explosion by not generating a separate combination of tuples for every way in a conjunction of conditions can be matched. First, it presents and discusses the key idea behind the approach. It then presents a new match algorithm based on this approach.

The tuple-oriented nature of the existing match algorithms is closely tied to the tuple-oriented semantics of the languages they are used to implement. Chapter 7 discusses language semantics and programming styles supported by the new class of match algorithms chapter. It presents the design and implementation of a collection-oriented production language, COPL.

Chapter 8 describes the experiments conducted to evaluate the scalability of collection-oriented match algorithms and to compare the performance of collection-oriented match algorithms and their tuple-oriented analogues for large tuple-spaces. In these experiments, Rete was selected as the exemplar for tuple-oriented match algorithms and its collection-oriented analogue, Collection Rete, was selected as the exemplar for collection-oriented match algorithms. The first section describes the benchmark programs. The next section describes the structure of the experiments. The third section compares the performance of PPL and COPL on the benchmark programs. The fourth section discusses the scalability of collection-oriented match algorithms. The chapter concludes with some observations from the experiments including programming idioms for collection-oriented production languages.

Chapter 9 discusses related work. It covers a broad range of research into parallel production systems ranging from the early efforts to parallelize production match on a variety of architectures (tree-structured architectures, dataflow machines, bus-based shared memory machines, SIMD machines and low latency message passing architectures) to compile-time and run-time analyses to determine which instantiations can be fired in parallel and parallel production system languages. It also discusses several efforts to eliminate or reduce the combinatorial explosion in the number of instantiations and partial matches. This includes efforts such as *unique-attributes* and *instantiation-less match* that are able to guarantee a polynomial bound on the number of instantiations and partial matches as well as less comprehensive schemes, such as *copy-and-constrain* and *data partitioning*, which are unable to provide such a guarantee but, in some cases, are able to reduce the total number of comparisons performed.

Finally, Chapter 10 presents the primary conclusions of this dissertation and discusses some avenues for future research.

Chapter 2

Background

This chapter provides the background necessary for the rest of the dissertation. The first section provides a brief introduction to the production system paradigm. The second section describes the syntax and semantics of OPS5. OPS5 is the archetypical production system language. It was the first production system language that saw wide use. Most later languages retained its basic structure. All the test programs used in this investigation were originally written in OPS5. The third section briefly describes efficient match algorithms and goes into more detail for one of them, Rete. Rete is the most commonly used match algorithm and to the best of this author's knowledge, is used in all publicly available efficient implementations of production systems. Several schemes have been suggested for parallelizing Rete. The fourth section describes the most successful scheme.

2.1 Production Systems

A production system program consists of a set of *if-then* rules (productions) and a tuple-space. The productions are the code for the program and the tuple-space contains the data being processed by the program. Each production consists of a pattern as the *if*-part, and a sequence of actions as the *then*-part. The patterns match tuples from the tuple-space and the actions modify the tuple-space. Execution of a production system program consists of a sequence of *match-select-act (msa)* cycles. In the *match* phase of each cycle, the patterns for all the productions are matched against the tuple-space. In the *select* phase, a subset of the matched patterns are chosen (the particular selection algorithm depends on the language being used) and the corresponding actions are executed. This usually results in modification of the tuple-space. The cycle then repeats with the new tuple-space. If, at any stage, the tuple-space fails to match even one of the patterns, execution is terminated. Figure 2.1 shows a high-level view of a

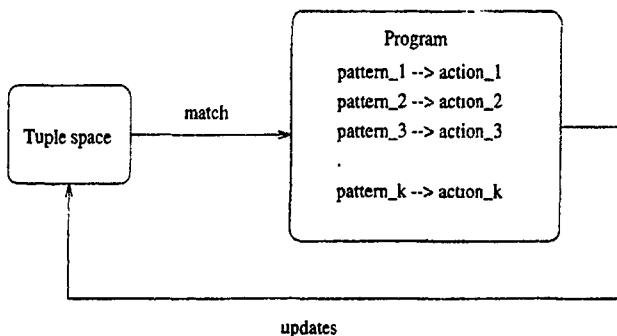


Figure 2.1: High-level view of a production system program

production system program. The key feature of the production system computational model is the highly data-dependent nature of the control flow.

Productions can be considered as daemons that watch the tuple-space for particular patterns. This view of productions is prevalent in the active database community [13, 24, 46, 108, 127]. They can also be viewed as guarded commands[22], though almost always without the nondeterminism. Most production system applications are based on this view. Finally, productions can be viewed as associative access mechanisms. Several prominent cognitive models use productions to model the associative nature of human memory [3, 66]. For more details on the production system computational model, see [125].

2.2 OPS5

OPS5 was the first widely used production system language [28]. It was designed as a part of the Instructable Production System project at Carnegie Mellon University and a Franzlisp implementation was developed by Charles Forgy. This implementation has been the basis for most subsequent implementations of OPS5 and its derivative languages. Most later production system languages retained the basic structure of OPS5 and extended it in various ways. Almost all recent research in implementation of production systems, sequential as well as parallel, has been within the context of OPS5.

The tuple-space for an OPS5 program is called the *working memory* and individual tuples are referred to as *working memory elements*. A tuple consists of a type tag (its *class*), a unique identifier (its *metag*) and a set of named fields. Field names are symbolic and are referred to as *attributes*. Field values can only be of ground types (integer, symbol or floating point).

Figure 2.2 shows an OPS5 tuple of class `student`, with four fields – `name`, `roll-number`, `score` and `course`. Field names are prefixed with “`^`” to distinguish them from symbolic values. The timetag associated with this tuple is 4.

```
4: (student ^name Bovik ^roll-number 142 ^score 93 ^course prog-language)
```

Figure 2.2: Sample OPS5 tuple

The if-part of an OPS5 production consists of a conjunction of conditions. Each condition is a pattern which is syntactically similar to a tuple and tests for the presence of tuples matching the pattern. A condition can be *negated*, that is, prefixed with “`-`”, in which case it tests for the *absence* of tuples matching the pattern. If the tuple-space contains a matching tuple for every non-negated condition in a production and no matching tuples for any of the negated conditions, it is said to match, or be *instantiated*. An instantiation of a production consists of the list of tuples which match all its non-negated conditions. The then-part of a production consists of a sequence of actions which may modify the tuple-space, perform IO or call functions in other languages. Figure 2.3 shows an OPS5 production with three conditions and two actions.

```
(p find-max-scoring-student-in-prog-language
  (student ^roll-number <rollno> ^name <name> ^score <score> ^course prog-language )
  -(student ^score > <score> ^course prog-language )
  (advisor ^name <name2> ^advisee <rollno> ^dept computer-science )
  ->
  (write Top score <score> by <name>, advised by <name2>)
  (make top-score ^score <score> ^course prog-language))
```

Figure 2.3: Sample OPS5 production

The condition patterns look like tuples but they may have *variables* in place of a field value. Variables are specified by symbols enclosed in angle brackets (`<>`), for example `<rollno>`. Each pattern specifies a set of tests for a tuple. Tests can be equality or relational. A constant field in a pattern can be matched only by tuples that have the same value for the same field. Such tests are referred to as *constant* tests. In Figure 2.3, the constant tests are in boldface. For example, the first condition has two constant tests, the first which checks if the tuple is of type `student` and the second which tests if the value of the `course` field for the tuple is `prog-language`. A variable field in a pattern can be matched by a tuple with *any* value for the field. If a variable occurs multiple times in a production, it must match the same value everywhere.

```
((student ^name Bovik ^roll-number 17 ^score 97 ^course prog-language)
 null
 (advisor ^name Shanti ^advisee 17 ^dept computer-science) )
<rollno> = 17, <name> = Bovik, <score> = 97, <name2> = Shanti
```

Figure 2.4: An instantiation for the sample production

In effect, variables are used to specify constraints or consistency checks between tuples that match different conditions. These tests are referred to as *variable* tests. For example, the first condition in Figure 2.3 has three variable tests which bind the variables *<rollno>*, *<name>* and *<score>* to the corresponding values in the matching tuple. The variable *<rollno>* occurs again in the third condition where it is used to specify that the value of the *advisee* field of the tuples matching this condition must be the same as the value of the *roll-number* field of the tuple matching the first condition. In practice, variables occurring in multiple conditions are often used to match an aggregate data object implemented as several tuples. For example, the production in Figure 2.3 uses the *<rollno>* variable common between the first and third conditions to link the students in the programming languages course to their advisors in the computer science department. To understand the matching process for a complete production, consider, once again, Figure 2.3. The first condition is satisfied if there exists a student who is taking the programming languages course. The second condition is satisfied if there exists *no* student in the programming language course whose score is higher than the score of the student matching the first condition. In other words, the student matching the first condition is the top-scoring student in the programming languages course. The inter-condition constraint is specified by the common variable, that is, *<score>*. The third condition is satisfied if there exists an advisor in the computer science department for the student matching the first condition. The variable *<name>* is bound to the name of the top-scoring student and the variable *<name2>* is bound to the name of her advisor. An instantiation for this production consists of the tuples matching the first and third conditions (in order). Figure 2.4 shows an example along with the bindings for the variables occurring in the production.

The set of all instantiations is referred to as the *conflict set*. The select phase uses a deterministic algorithm to choose at most one instantiation from the conflict set. It first tries to order the conflict set using the age of the tuples contained in the instantiations (*recency* criterion). Instantiations containing younger tuples are preferred over instantiations containing older tuples. If this does not yield a total order, it tries to break the ties using syntactic measures like the number of conditions and tests (the *specificity* criterion). If this fails, it imposes an arbitrary order on instantiations that are still tied. It then picks the dominant instantiation from the total

order.

In the act phase, the selected instantiation is *fired*, that is the actions in the corresponding production are executed after replacing the occurrences of variables by the values bound to them. For example, the instantiation in Figure 2.4 is fired by executing the sequence:

```
(write Top score 97 by Bovik, advised by Shanti)  
(make top-score ^score 97 ^course prog-language)
```

This results in "*Top score 97 by Bovik, advised by Shanti*" being written on the output stream and the tuple `(top-score ^score 97 ^course prog-language)` being added to the tuple-space. Timetags for all tuples are automatically assigned by the implementation.

For further details on OPSS, see [11].

2.3 Match Algorithms

The match phase has traditionally been the most expensive segment of production system execution and therefore has been the focus of much research [27, 29, 30, 39, 45, 78, 79, 90, 94]. Early measurements indicated that, on the average, the rate of change of tuple-space is low [27] (this is referred to as *temporal redundancy*). As a result, all known efficient match algorithms are incremental, that is, in each *msa* cycle, they match only the changes to the tuple-space instead of matching the entire tuple-space. This is achieved by preserving the state of the match algorithm across *msa* cycles. The amount of state preserved depends on the particular match algorithm being used. It could be as little as keeping track of the sets of tuples matching individual conditions and as much as keeping track of all possible matches for all possible sequences of conditions. Besides efficiency, there is also a semantic reason for preserving information about matches across *msa* cycles. Production system semantics dictate that an instantiation can be fired at most once. The purpose of this restriction is to avoid infinite loops due to the same instantiation being fired over and over again. Therefore, it is necessary to keep track of the set of instantiations that have already been fired. Such instantiations are referred to as *refracted* instantiations. Practically, this means the conflict set has to be preserved across *msa* cycles.

The primary difference among the major match algorithms in the literature is the amount of state they save. At one end of the spectrum is the Dynamic Join algorithm [90] which stores information about matches for all possible combinations of conditions. For a production with three conditions C1, C2 and C3, Dynamic Join stores information about the individual tuples that match C1, C2 and C3; the tuple-pairs that match {C1,C2}, {C1, C3} and {C2, C3} and the tuple-triples that match {C1, C2, C3}. The state saved by this algorithm grows very

rapidly with the increase in the size of the tuple-space. Furthermore, most of the state contains redundant information. Therefore, it is not suitable as a base for a scalable implementation.

At the other end of the state-saving spectrum are Treat [78] and its derivative algorithms [44, 79]. Treat stores information only about the matches for individual conditions and for complete productions. For a production with three conditions, Treat would store information about the tuples that match C1, C2 and C3 and the tuple-triples that match {C1,C2,C3} [78]. Variations of Treat store even less information: Lazy Match [79] stores only one instantiation per production and A'-Treat [44] does not store the matches for individual conditions. These algorithms recompute the rest of the state as and when needed. They are suitable for programs whose tuple-spaces change fast enough to make state-saving not worthwhile. For most of the applications being considered in this dissertation, the tuple-space changes very slowly (e.g., databases, images etc.). Therefore, these algorithms would incur substantial recomputing costs which could be avoided by additional state-saving.

Rete [29] and its derivative algorithms [38, 100, 114] lie between these extremes. In addition to storing information about the tuples matching individual conditions, Rete stores information about tuples that match a *particular sequence* of combinations which is fixed in advance. For a three condition production, Rete stores information about the tuples that match C1, C2 and C3; the tuple-pairs that match {C1,C2} and the tuple-triples that match {C1,C2,C3}. The sequence of conditions can be picked to minimize the size of state to be saved. Since it saves more state than Treat and its derivatives, Rete is more suitable for slowly changing tuple-spaces. As noted in the previous paragraph, in most applications under consideration, the tuple-space changes slowly. Therefore, Rete appears to be the algorithm of choice.

However, in choosing the algorithm for a scalable implementation, the savings in the recomputation cost have to be traded off against the increase in the space requirement and against the limitations each choice imposes on the semantics of the language. Since Rete, Treat and A'-Treat store the conflict set, their worst case space complexity is $O(w^n)$ where w is the size of the tuple-space and n is the length of the longest pattern. Lazy Match reduces the worst-case complexity by storing only one instantiation per production. This means it can be used only for languages with sequential semantics. As discussed in Chapters 5 and 7, this is undesirable for programs with large data sets. UniRete [114] restricts both the tests that can appear in conditions and the values that can appear in the tuples to limit the number of tuples that can match individual conditions to at most one. This reduces its space complexity to $O(n)$ per production. However, these semantic restrictions may necessitate a much larger number (sometimes exponential number) of productions to achieve the same functionality [112]. A'-Treat [44] tries to reduce the average case complexity by not storing the tuples matching individual conditions for conditions which are expected to match a large number of tuples. This is effective if consistency checks on these conditions are infrequent. However, the need for consistency checks is data-dependent and cannot be predicted in the general case.

In addition to being suitable for the slowly-changing tuple-spaces of the application areas of

interest, Rete supports succinct programs in sequential as well as parallel computational models. Therefore, Rete has been chosen as the base of this investigation. Note however, that if the amount of match state does not fit in memory, it may be necessary to spill part of the state to disk. This can be done either on a per-page basis (as in commonly used virtual memory systems) or on a per-object basis, objects in this case being the match state for individual conditions or condition sequences. Another possibility is the use of operating system support for *recomputable pages* [109] which would not spill the state to disk but would recompute it as and when necessary.

2.4 Rete

Rete was developed by Charles Forgy for the implementation of the OPS family of production system languages. It has since become the most commonly used match algorithm for production systems. As mentioned in the previous section, Rete is suitable for matching against slowly-changing tuple-spaces. In addition to its position on the state-saving spectrum, it is also characterized by the fixed order in which the tests are performed. In contrast, Treat and Dynamic Join change the order in which conditions are tested at run-time.

Rete is based on a dataflow network generated from the conditions of the productions. For example, consider the productions in Figure 2.5. The Rete network corresponding to these productions can be found in Figure 2.6. The network can be divided into two parts, the upper half which implements the intra-condition tests and the lower half which implements the inter-condition tests.

The upper half of the Rete network is referred to as the α network and consists of a discrimination net based on individual conditions. For any tuple, the α network determines the set of conditions it matches. The discrimination net exploits similarity between conditions (within and across productions) by sharing the tests for identical conditions. For example, in Figure 2.5, the first condition in both productions checks if the type of the tuple is xor-gate and the rest of the conditions in both productions check if the type is line. There are no other intra-condition tests. Accordingly, there are only two branches in the α network in Figure 2.6.

The lower half is referred to as the β network and performs the inter-condition consistency checks. The β network consists of β nodes which perform consistency checks between tuples. In Figure 2.6, β nodes are represented by circles. The tests performed by an individual node are specified on its left. This network exploits similarity between productions by sharing network segments for common condition prefixes. The consistency test between the first two conditions for both productions in Figure 2.5 is identical, it checks if the value of the in1 field of the tuple matching the first condition is same as the value of the id field of the tuple matching the second condition. In the network, this test is implemented by β_1 and is shared between both the productions. Since the next condition, the third, is different for the two productions, there

```

(p xor-gate-on
  (xor-gate ^in1 <input1> ^in2 <input2> ^out <output>)
  (line ^id <input1> ^value <v>)
  (line ^id <input2> ^value <v>)
  (line ^id <output>)
  ->
  (modify 4 ^value 0))

(p xor-gate-off
  (xor-gate ^in1 <input1> ^in2 <input2> ^out <output>)
  (line ^id <input1> ^value <v>)
  (line ^id <input2> ^value <> <v>)
  (line ^id <output>)
  ->
  (modify 4 ^value 1))

```

Figure 2.5: Productions to implement a simulator for 2-input xor-gates

is a separate subnetwork for the rest of the conditions of each production. The subnetwork for xor-gate-off contains β_2 and β_4 which correspond to its third and fourth conditions. Similarly, the subnetwork for xor-gate-on consists of β_3 and β_5 . There are two kinds of β nodes – *and* nodes and *not* nodes. The former perform tests corresponding to non-negated conditions whereas the latter perform tests corresponding to negated conditions. All the nodes in Figure 2.6 are *and* nodes.

The objects that flow down the Rete network are tuple combinations. They are referred to as *tokens*. Tokens correspond to matches for condition prefixes and consist of an ordered sequence of tuples, one tuple corresponding to each non-negated condition and a null tuple for each negated condition. To preserve information about partial matches, *memory nodes* are inserted in the Rete network. Memory nodes are needed at the bottom of the discrimination net (to preserve information about matches for individual conditions) and after every β node (to preserve information about matches for condition prefixes). In Figure 2.6, memory nodes are depicted by rounded rectangles. With the insertion of the memory nodes, each input of a β node comes from a memory node. The memory node on the left input of a β node is referred to its *left memory* and the memory node on the right input of a β node is referred to its *right memory*. Memory nodes at the bottom of the network are special as they contain complete

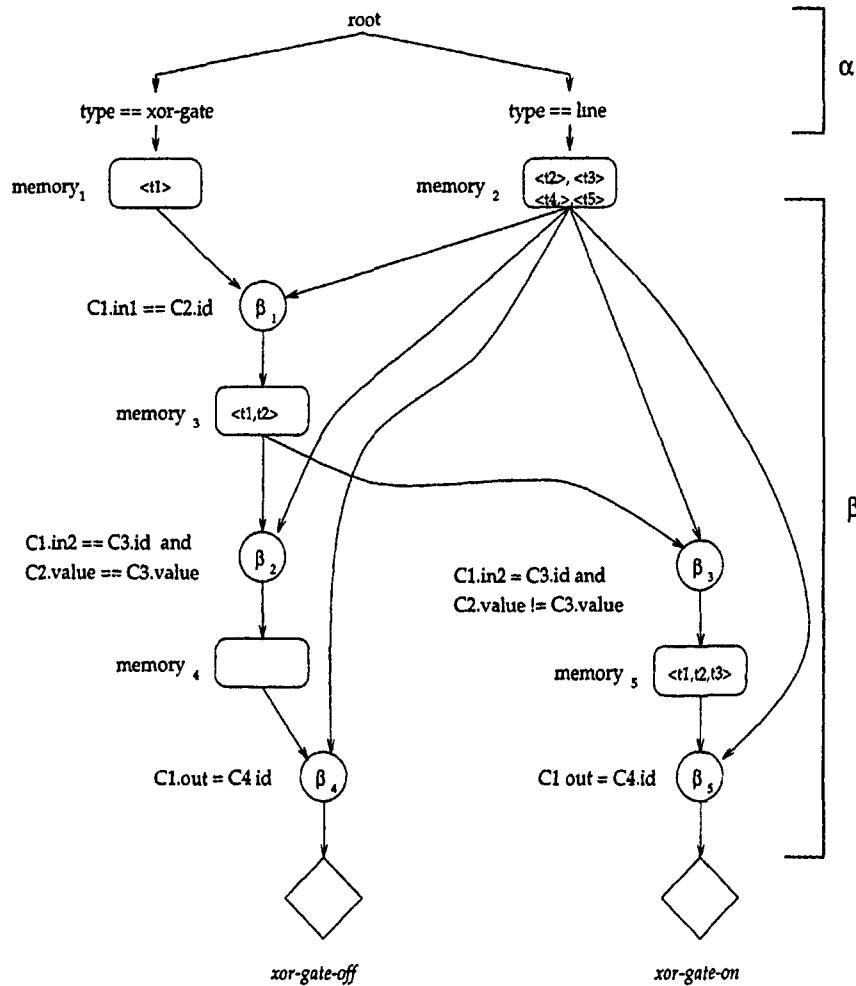


Figure 2.6: Rete network for the XOR gate productions

and not partial matches. They are referred to as *terminal* nodes or *pnodes* and are depicted as diamonds in Figure 2.6. The contents of all terminal nodes constitutes the conflict set.

To illustrate the operation of the algorithm, consider the network in Figure 2.6 and the following sequence of additions to the tuple-space.

```
t1: (xor-gate ^in1 line1 ^in2 line2 ^out line3)
t2: (line ^id line1 ^value 0)
t3: (line ^id line2 ^value 1)
t4: (line ^id line3 ^value 1)
```

When the first tuple, t_1 , is inserted into the tuple-space, the token $\langle t_1 \rangle$ is created and sent to the root node of the network. The root node broadcasts it on all the branches emanating from it. Since, its type is *xor-gate*, only the test on the left branch succeeds. The token flows into *memory*₁ which stores it and passes a copy to its successor, β_1 . β_1 compares this token with the tokens in its right memory. The right memory is currently empty, so there are no matches and no further propagation takes place.

When the second tuple, t_2 , is inserted into the tuple-space, the token $\langle t_2 \rangle$ is created and broadcasted from the root node. Since its type is *line*, only the test on the right branch succeeds. It gets stored in *memory*₂ and a copy is passed to the successor β nodes, which, in this case, is all the beta nodes in the network. Each node checks the corresponding opposite (left) memory for matches. Only β_1 finds a matching token, $\langle t_1 \rangle$, in its left memory (*memory*₁), since the *id* field of t_2 has the same value as the *in1* field of t_1 . This results in the generation of the successor token $\langle t_1, t_2 \rangle$ which is stored in *memory*₃ and a copy passed to β_2 and β_3 . Both these nodes check *memory*₂ for a tuple whose *id* field has the same value as the *in2* field of t_1 . Since no such tuple exists (yet), no further propagation takes place.

When the third tuple, t_3 , is inserted into the tuple-space, the token $\langle t_3 \rangle$ is created and broadcast from the root node. Again, only the test on the right branch succeeds causing the token to be added to *memory*₂ and copies passed to all the β nodes. Only β_4 finds a matching token, $\langle t_1, t_2 \rangle$, in its left memory (*memory*₃), since the *id* field of t_3 contains the same value as the *in2* field of t_1 and the *value* fields of t_2 and t_3 are different (0 and 1 respectively). This leads to the generation of $\langle t_1, t_2, t_3 \rangle$ which is stored in *memory*₅ and a copy passed to β_5 . Since the right memory for β_5 does not (yet) contain a tuple whose *id* field is the same as the *out* field of t_1 , no further propagation takes place.

When the fourth tuple, t_4 , is inserted into the tuple-space, the token $\langle t_4 \rangle$ is created and broadcast from the root node. Once again, only the test on the right branch succeeds causing the token to be added to *memory*₂ and copies passed to all successor β nodes. Only β_5 finds a matching token, $\langle t_1, t_2, t_3 \rangle$, in its left memory (*memory*₅), since the *id* field of t_4 contains the same value as the *out* field of t_1 . This leads to the generation of the complete match

```

{ (xor-gate ^in1 line1 ^in2 line2 ^out line3)
  (line ^id line1 ^value 0)
  (line ^id line 2 ^value 1)
  (line ^id line3 ^value 1) }

<input1> = line1, <input2> = line2, <output> = line3, <v> = 0

```

Figure 2.7: Instantiation generated for the xor-gate simulator

t_1, t_2, t_3, t_4 which flows into the terminal node for the production xor-gate-on. The instantiation generated is shown in Figure 2.7.

Rete handles the deletion of tuples similarly except that matching tokens are deleted from memory nodes instead of being added. Rete adds a tag to every token indicating whether it corresponds to an addition or a deletion. Modification of tuples is implemented by deletion of the old tuple followed by addition of the updated tuple.

For further details on the Rete algorithm see [29].

2.5 Parallel Rete

The key feature of the Rete algorithm which makes it possible to parallelize it is its dataflow nature. It provides a natural way of decomposing the task of matching of a set of rules into small subtasks which can be done in parallel. Several schemes have been suggested for parallelizing Rete [2, 37, 38, 40, 51, 61]. This section describes the most successful scheme. This scheme has been used in ParaOPSS [42] and CParaOPSS [1], which are parallelizing implementations of OPSS on shared memory machines. It is also used in the implementations developed for this dissertation.

Individual tests in the α network are too small (between three and nine instructions) to be scheduled as separate tasks. All the tests performed on a token as it flows through the α network is a more suitable task unit. Processing of a token at a β node is more substantial (usually between one and seven hundred instructions) and can be considered as a potential task unit on its own.

The intuitive scheme for parallelizing Rete arises from viewing it in an *object-oriented* manner where the nodes of the Rete network are the objects and the tokens are the messages. In this scheme, the nodes of the Rete network are partitioned and allocated to the available processors. The primary problem for this scheme is the irregular and data-dependent nature of computation

which makes it difficult to determine good partitioning strategies. This leads to poor load balancing and processor utilization. Dynamic migration of nodes between partitions is not feasible since the activity in the Rete network is highly irregular and knowledge of past activity does not help in predicting loci of future activity [2]. Furthermore, the number of tokens flowing into individual nodes can often be quite large. Since each node is typically allocated to a single processor, this leads to serialization of the processing of all tokens flowing into a single node. Replicating β nodes is not a feasible solution for this problem since a token flowing into a β node has to be compared with *all* tokens in the opposite memory. All tokens flowing into a node would have to be broadcasted to and stored by all replicas. Furthermore, the unpredictable nature of the token flow in the network makes it difficult to determine an appropriate degree of replication.

To avoid these problems, Anoop Gupta [38] suggested that all processors should be permitted to process tokens destined for all nodes. For scheduling and load balancing, he suggested the use of shared task pools. To avoid serializing on access to a memory node on an input of a β node, he suggested that contents of the memory nodes be partitioned. Since his measurements indicated that a large number of memory nodes are empty, he recommended that two global hash-tables be used to store the contents of all memory nodes – one for all left memories and the other for all right memories. The hash function takes the identifier of the destination β node of the token and the value(s) being tested at the node as parameters. Using the value(s) being tested as parameters implies that the tokens that flow into the same node but have different values for the tested fields get hashed to different buckets which can be accessed in parallel. As long as the value(s) being tested are different, this scheme avoids serializing on tokens destined for the same node. Similarly, using the identifier of the destination β node as a parameter results in distribution of tokens flowing into different β nodes to different hash buckets, allowing them to be processed in parallel. Furthermore, hashing the contents of the memory nodes, instead of storing them in linear lists reduces the average number of comparisons performed per token [38]. Since Rete spends most of its time in the β network, using hashtable based memory nodes also improves uniprocessor performance.

The use of a hashtable, however, implies that most memory nodes cannot be shared. This is because the hash function uses the value(s) being tested at the destination β node and different successor nodes test different fields (or else they would have been shared). However, it is still possible to share β nodes in which case the associated memory nodes get automatically shared.

While processing changes to the tuple-space, it is possible that the same token is first added to and then deleted from a memory node. Such token-pairs are referred to as *conjugate* tokens [27]. Since conjugate tokens can be processed by different processors, the deletion request may precede the corresponding addition request at the memory node. For correctness, the deletion request has to be held at the memory node until the addition request arrives. This implies that some sort of an *extra-deletes-list* [38] has to be associated with every memory node (including the terminal nodes). For a hashtable based implementation of memory nodes,

processing of conjugate tokens can be integrated with that of normal tokens by providing *conjugate* hashtables which mirror the normal hashtables. The operation of adding a token to a hash bucket is modified to include a check of the corresponding bucket in the conjugate hashtable.

For further information on parallel implementations of Rete, see [38].

Chapter 3

Design and Implementation of PPL

In the most attractive parallel programming scenario, the programmer writes a sequential program which is correct and efficient and the compiler takes care of parallelizing it. There has been considerable research on automatic parallelization of production systems. However, all of the efforts have had limited success. The primary reason for this is the data-dependent nature of control flow in production system programs. Since compilers do not have information about the run-time contents of the tuple-space, they are forced to be overly conservative in their efforts to determine which operations can be safely performed in parallel. This chapter discusses and evaluates the ways in which information about the run-time contents of the tuple-space can be made available to a parallel implementation and how this information can be utilized.

The first section describes the limitations of automatic parallelization for production system programs and establishes the need for parallel production system languages. The second section lays out the design space for these languages and evaluates the alternatives. The third section describes the design of a particular language, Parallel Production Language (PPL). The fourth section describes the PPL implementation.

3.1 Need for Parallel Production Languages

This section argues that automatic parallelization of sequential production system languages is subject to a program-independent bound on the average number of tasks that can be performed in parallel (the *available parallelism*) and that parallel production system languages are needed for scalable parallelism. The primary cause of this program-independent bound is the uniformly high frequency of barrier synchronizations in parallel execution of production system programs. Since there is little work to be done between successive barrier synchronizations, only a small number of processors can be gainfully employed. Section 3.1.1 shows that the semantics of production system languages require a barrier synchronization in every *msa* cycle. Section 3.1.2

discusses why barrier synchronizations can be expected to be frequent. Eliminating the limit on the amount of work between successive barrier synchronizations is necessary for scalable parallelism but not sufficient. It is also necessary to ensure that the available parallelism keeps pace with the growth in the amount of work. Section 3.1.3 discusses the desiderata for a parallel production system language.

3.1.1 Need for barrier synchronization

The production system computational model is *synchronous*. In each match-select-act cycle, the match phase must be completed before the instantiations to be fired can be selected. Parallel implementations of production system programs need a barrier synchronization to ensure this. There are three reasons for this requirement:

Negation as non-existence: Recall from Section 2.2 that the if-part of a production can contain negated conditions. A production containing negated conditions is considered to have a match if and only if the following assertions hold simultaneously:

1. For each non-negated condition, the tuple-space contains a tuple that satisfies its constraints
2. For each negated condition, the tuple-space does not contain a tuple that satisfies its constraints

The first assertion can be shown to hold as soon as a matching tuple has been found for every non-negated condition but the truth of the second assertion can be established only after the entire tuple-space has been processed. For incremental match algorithms this means matching negated conditions is completed only when all changes to the tuple-space have been processed.

Global selection policies: Non-trivial selection algorithms, that is, algorithms that pick a proper subset of the conflict set, usually pick the "best" subset for some measure of quality (the idea being to execute the most appropriate rule applicable). To be able to do this, it must have access to *all* instantiations which implies that the final selection can take place only after the match phase has been completed. It is possible to integrate the algorithm that selects the instantiations with the algorithm that generates the instantiations such that only the desired subset of the instantiations is generated. Or that the instantiations are generated in decreasing order of desirability. For example, the Lazy Match algorithm[80] integrates the LEX selection algorithm [11] with the Treat match algorithm to avoid generating instantiations that will not be selected. In the absence of negated conditions, such a strategy would allow instantiations to be fired as soon as they are generated. However, this is possible only for simple selection algorithms that depend on structural features like timestamps on tuples and size of instantiations. Selection strategies that depend on the contents of the instantiations, like meta-rules in [18]

and `redact` rules in PARULEL programs [107] must, in general, await the generation of all instantiations before making decisions.

Non-monotonic tuple-space: Previous paragraphs seem to indicate that production system programs with no negated conditions and structural selection algorithms can run without synchronization points. This is true if and only if there is no *interference* between instantiations. Two instantiations interfere if firing one of them deletes tuples contained in the other or if both try to modify the same tuple.¹ As long as deletion or modification of tuples is permitted, non-interference of all possible instantiations can be guaranteed only for very simple programs. For example, consider the production in Figure 3.1. This production interferes with itself since the two instantiations generated for the given tuples try to paint the same red ball. For non-monotonic tuple-spaces, asynchronous firings would lead to non-deterministic behavior and race conditions. Monotonicity of the tuple-space is a severe restriction since either the data must not evolve or multiple copies of the data must be maintained. Monotonic tuple-spaces occur in "pure" Prolog programs [17]. Subsequent logic programming languages have introduced non-monotonic tuple-spaces [9, 69, 85].

```
(p paint-red-ball
  (ball ^color blue)
  (ball ^color red)
  -->
  (modify 2 ^color blue))
```

```
T1: (ball ^color blue)
T2: (ball ^color blue)
T3: (ball ^color red)
```

Instantiations: <T1,T3>, <T2,T3>

Figure 3.1: Simple production that interferes with itself

¹This definition assumes that there are no negated conditions in the program. It can be easily extended for programs with negated conditions.

3.1.2 Frequency of barrier synchronization

There are two levels at which a sequential production system program can be parallelized – parallel execution of operations within individual *msa* cycles and parallel/overlapped execution of multiple *msa* cycles. The following subsections argue that, in both cases, the frequency of barrier synchronizations can be expected to be high.

3.1.2.1 Intra-cycle parallelization

For parallel execution, the match-select-act cycle is converted to the match-barrier-select-act cycle. In other words, parallel execution of production system programs consists of a sequence of select-act-match cycles separated by barrier synchronizations. The total cost of operations between successive synchronizations is a sum of:

1. **Selection cost:** depends on the number of changes to the conflict set (i.e., the number of instantiations generated or deleted) during a cycle and on the number of instantiations selected
2. **Firing cost:** depends on the number of instantiations selected and the costs of firing individual instantiations.
3. **Match cost:** depends on the number of modifications to the tuple-space and the number of productions that can potentially match each modified tuple. (Recall from Chapter 2 that efficient match algorithms are incremental and match only the changes to the tuple-space in each cycle.)

The average number of changes to the conflict set in a cycle is usually small. For the programs studied in Anoop Gupta's thesis[38], the average number of conflict set modifications ranges between 2.5 and 7.86, the weighted average being 4.57. A previous study by Gupta and Forgy[39], on a different set of programs, measured the average number of conflict set modifications to be between 3.2 and 12.6 and sequential production system languages allow only one instantiation to be fired every *msa* cycle. As a result, the selection cost is small. Since each production has a small number of actions (≈ 3 [38]) and since individual actions are cheap (≈ 200 instructions), the firing cost is usually small. Furthermore, since the number of tuple-space modifications per *msa* cycle are small (≈ 2.51 [38]) and since the number of productions that can match individual modifications is usually small (≈ 32 [38]), the amount of time spent in the match phase of each cycle is usually small.²

²This does not hold for some cases. For example, the addition of a single *sequencing* tuple can cause a large number of instantiations to be generated. However, such cases are relatively rare.

Since the work being done in each phase is usually small, the total work being done between successive barrier synchronizations is small and the frequency of the barrier synchronizations can be expected to be high. Gupta *et al.* [42] report that a large fraction of the *msa* cycles in a suite of OPSS programs had fewer than 250 tasks (each task being between 100 and 500 instructions). They referred to these cycles as *short cycles* and concluded that the preponderance of these cycles was responsible for the low speedup achieved. Tambe *et al.* [113] report similar results for a parallel implementation of Soar [66].

The exceptions to this are production system programs that call expensive foreign functions. For example, the SPAM image-interpretation system [75] calls geometric functions to compute relationships between image regions; these functions are expensive to compute and dominate the computation in SPAM. Even though the frequency of barrier synchronizations is not high in such programs, the parallelism available in individual *msa* cycles remains low since the foreign functions used are usually written in a sequential language and are not available to the production system compiler.

3.1.2.2 Inter-cycle parallelization

Since a barrier synchronization occurs in every *msa* cycle, the only way to go beyond the limitations imposed by sequential language semantics is to combine multiple *msa* cycles and share a single barrier synchronization between them. Figure 3.2 shows an example for three cycles. In such composite cycles, the match phase determines the instantiations, the select phase picks multiple instantiations to fire and the act phase executes the actions corresponding to all of them. To preserve sequential semantics, a set of cycles can be combined if and only if the implementation is able to prove that there are no dependencies between the instantiations being fired in these cycles. There can be two kinds of dependencies between a pair of instantiations, *read-write* and *write-write*. An instantiation is said to *read* a tuple if it contains the tuple.³ An instantiation is said to *write* a tuple if firing it causes the tuple to be deleted or modified. A *read-write* dependency occurs between two instantiations if one of them writes a tuple read by the other and a *write-write* dependency occurs between two instantiations if both of them write the same tuple. For example, consider the productions in the xor-gate simulator shown in Figure 3.3. Since instantiations of *xor-gate-on* write tuples of type *line* and instantiations of *xor-gate-off* read tuples of type *line* and vice-versa, there is a bidirectional *read-write* dependency between the two productions. That is, firing of an instantiation of *xor-gate-on* can potentially modify a tuple that is currently matching an instantiation of *xor-gate-off* and vice-versa. Furthermore, since both productions modify tuples of type *line*, there is a *write-write* dependency between them.

³Strictly speaking, this definition holds only for instantiations of productions with no negated conditions. It has been simplified for illustration and can easily be extended to full generality.

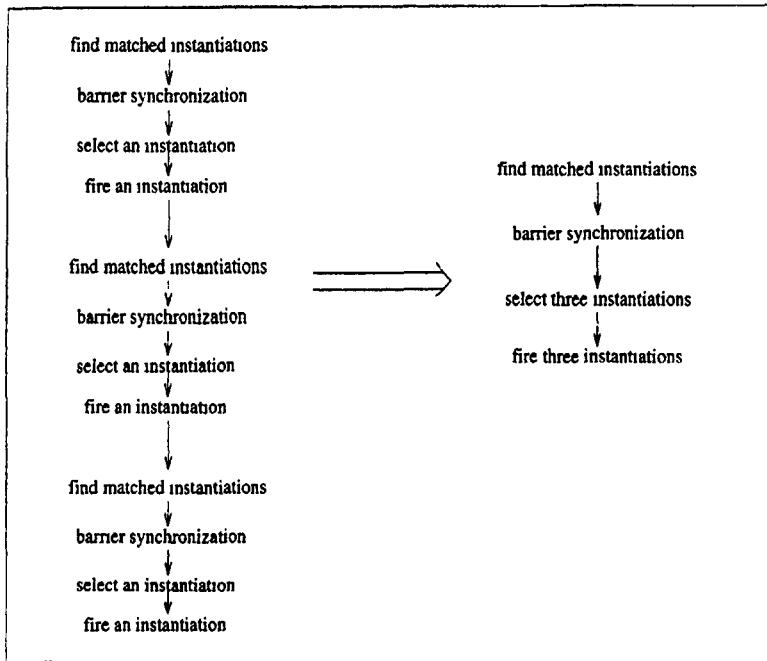


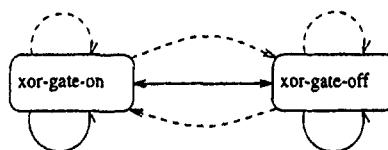
Figure 3.2: Execution of three match-select-act cycles in parallel

Ishida and Stolfo [56] proposed a compile-time analysis, based on a dependency-graph, to determine which cycles could be safely combined. The nodes in their dependency-graph correspond to individual productions and the links represent the dependency information available at compile-time. There is a link between two nodes, A and B, if there is a dependency between any pair of instantiations of the corresponding productions. If the actions of production A modify a tuple of the same type as one of the tuples matched by production B, the graph has a directed *read-write* link between A and B. If the actions of two productions modify a tuple of the same type, the graph has a bidirectional *write-write* link between the two. Figure 3.3 shows the dependency-graph for the xor-gate simulator. In addition to the direct dependencies represented by the links, two productions can have an indirect dependency between them if one of them lies in the transitive closure (over the dependency links) of the other. Two productions

are independent if neither lies in the transitive closure of the other. This analysis uses only the type of the tuples. Subsequent efforts by Tenorio and Moldovan[119], Miranker *et al.* [81] and Schmolze and Goel[101] have refined the analysis by taking advantage of the constant literals that occur in both the if-part and the then-part of the productions.

```
(p xor-gate-on
  (xor-gate ^in1 <input1> ^in2 <input2> ^out <output>)
  (line ^id <input1> ^value <>>)
  (line ^id <input2> ^value <>>)
  (line ^id <output>)
  -->
  (modify 4 ^value 0))

(p xor-gate-off
  (xor-gate ^in1 <input1> ^in2 <input2> ^out <output>)
  (line ^id <input1> ^value <>>)
  (line ^id <input2> ^value <>>)
  (line ^id <output>)
  -->
  (modify 4 ^value 1))
```



Dashed lines represent read-write dependencies and solid lines represent write-write dependencies

Figure 3.3: Dependency-graph for xor-gate simulator

Compile-time dependency analysis of production system programs is seriously limited by the data-dependent nature of the computation. Since the compiler has no information about the run-time contents of the tuple-space, it is forced to be overly conservative and often fails to prove the independence of productions that are obviously independent. For example, consider

the productions of the xor-gate simulator shown in Figure 3.3. Simulation of every xor-gate can be done in parallel. So, we would expect that all instantiations of xor-gate-on and xor-gate-off can be executed in parallel. However, a conservative analysis is unable to detect this as:

- both productions modify tuples of type line, therefore there is a write-write dependency between them and
- since both productions also match tuples of type line, there is a bidirectional read-write dependency between them.

This is not a limitation of this particular technique for compile-time analysis, rather of compile-time analysis itself. To be able to determine that xor-gate-on and xor-gate-off are independent, the compiler needs to prove that each line occurs on the output of one and only one xor-gate. In the absence of information about the run-time contents of the tuple-space, there is no way for a compiler to prove this.

Therefore it is not surprising that compile-time dependency analysis has had limited success. Several publications have claimed small constant factor reductions in the number of *msa* cycles [56, 81, 101, 119]. The stated assumption of these publications is that given enough processors, the time taken for each cycle would be the same. The unstated assumption is that all tasks that are generated from the firing of different instantiations can be performed independently. These assumptions are unsound as shown by the results presented in [2, 38, 42, 113]. These results indicate that inter-task dependencies are a major limitation on speedups in production system programs. Therefore, a measure based on the number of cycles is inherently flawed.

To work around the limitations of compile-time analysis, Oshisanwo and Dasiewicz suggested a run-time analysis of instantiations[91]. Their scheme inserts a dependency analysis phase between the match and select phases. This analysis uses the selection procedure of the sequential language to impose a total order on the instantiations currently in the conflict set. It then checks every instantiation for interference with instantiations that are above it in the ordering. If an instantiation does not interfere with any of the preceding instantiations, it is fired. A similar scheme has been proposed by Ishida[55].

Since the only information available to such schemes is the set of instantiations that are present in the conflict set during one cycle, they are able to detect only direct dependencies. This means they are able to identify the instantiations which modify tuples used to generate other instantiations in the conflict set. In the absence of information about future instantiations, they are unable to detect indirect dependencies. For example, suppose the conflict set in the first cycle has two non-interfering instantiations, A and B. A parallelizing implementation based on run-time analyses like those mentioned above would fire both A and B. Now suppose the firing of A leads to the generation of C which interferes with B, say, by deleting a tuple contained

in B. Under sequential semantics, it is possible that A is selected for firing in the first cycle and C in the second. In that case, B is deleted from the conflict set without being fired. It is possible to construct such examples for any non-trivial selection procedure (Figure 3.4 shows an example for the OPS5 selection procedure). Therefore, without some sort of lookahead, it is not possible to guarantee that the sequential and the parallelized versions of the program generate the same result. Adding lookahead to such schemes would, in general, require the analysis procedure to explore the space of possible execution paths. Since there is no limit on the depth of the lookahead that might be needed, such analyses could be prohibitively expensive even for the small conflict sets seen in [38].

(p A (class1))	(p B (class2))	(p C (class3))
-->	-->	-->
(make class3))	(make class4))	(delete 2))

Figure 3.4: OPS5 example of indirect dependency

Compile-time as well run-time dependency analysis was used by Kuo *et. al* in the CREL implementation on a 15 processor Sequent Symmetry [64]. In addition to performing dependency analysis to determine which instantiations can be fired in parallel, they modified the semantics of the OPS5 language to eliminate the recency feature from the selection procedure. They report speedups between two and six fold using up to 15 processors. They conclude that the primary reason for the low speedup was the programming style used which limited the ability of the implementation to prove independence between productions.

A commonly used idiom in production system programs is the use of a *context* tuple to direct the control-flow. Productions in programs that make use of this idiom test the context element; only the productions that test for the context currently in the tuple-space can possibly match. Kuo *et. al*[65] proposed a variation of the Ishida-Stolfo analysis where the entities being scheduled were *contexts* and not individual productions. The contexts that were proved to be independent by this analysis were executed in parallel. The primary limitation of this scheme is the fact that the programs it analyzes are written in a sequential language. The context-tuple idiom is used most often to enforce a particular execution path. This often results in spurious inter-context dependencies.

3.1.3 Desiderata for a parallel language

From the preceding discussion it is clear that the primary limitation on parallelism in production system programs is the program-independent bound on the amount of work to be done between successive barrier synchronizations. To support scalable parallelism, a production system language must eliminate this bound; it must allow the programmer to control the amount of work between barrier synchronizations. Eliminating this bound is necessary for scalable parallelism but not sufficient. It is also necessary to ensure that parallelism available between barrier synchronizations keeps pace with the growth in the amount of work.

As discussed in Section 3.1.2.2, the only way to increase the work between barrier synchronizations is to combine multiple *msa* cycles. To support scalable parallelism, it should be possible to combine an unbounded number of cycles in this manner. Section 3.1.2.2 shows that the lack of information about run-time contents of the tuple-space forces parallelizing implementations to be overly conservative. It is not necessary to completely specify the contents of the tuple-space, only certain characteristics of it. In the xor-gate simulator in Figure 3.3, it is the fact that every line appears on the output of only one xor-gate, that is, tuples corresponding to individual different lines can be modified independently. The language must allow the programmer to specify as much information about the contents of tuple-space as necessary.

The next section describes the design space of languages that satisfy this requirement and discusses the pros and cons of individual choices.

3.2 Design Space of Parallel Production Languages

The primary decision to be made in the design of a parallel production system language is the manner in which information about the contents of the tuple-space is to be provided. An obvious way to allow the programmer to specify this information would be to provide a data description sublanguage. Such a language could specify, for example, axioms about sets embedded in the flat tuple-space, their cardinality, their relationship with other sets and so forth. For the xor-gate simulator, the specification could take the form:

$$\begin{aligned} \forall (\text{xor-gate} \wedge \text{output } <\text{output}>), \text{cardinality}((\text{line} \wedge \text{id } <\text{output}>)) = 1 \wedge \\ \forall (\text{line} \wedge \text{id } <\text{line}>), \text{cardinality}((\text{xor-gate} \wedge \text{output } <\text{line}>)) \leq 1 \end{aligned}$$

which states that for each xor-gate, there is exactly one line on its output and that every line is on the output of at most one xor-gate (input lines are not on the output of any xor-gate). This would allow a compiler to infer that every tuple corresponding to a line can be updated independently.

The main advantage of this approach is its explicit nature. It makes explicit the assumptions with which the program is parallelized. At the cost of greatly slowing down the execution, it is

possible to check if these assumptions hold through out the execution. Such a checker would be a useful debugging tool.

However, this approach has two major disadvantages:

1. While it is easy to specify patterns in the data for small or regular tuple-spaces, doing so for large irregular tuple-spaces is likely to be difficult. This is particularly so for long-running programs whose data tends to evolve, like active databases, or where information about data values rather than data structure is needed for successful parallelization
2. A large number of patterns or relationships can usually be specified for large and complex data sets. To do a good job of specifying the necessary information, the programmer would need to have a good grasp of the compiler algorithms that use this information. This would hamper performance tuning.

The alternative approach is to specify which operations can be performed in parallel. In this approach, the assumptions (or the information) about the data are implicit in the specification. This is a lower-level specification as the programmer has to specify which operations can be done in parallel. However, the existence of an explicit operational model facilitates performance tuning. Since performance and ease of use are major concerns of this investigation, this approach is preferable to data specification. In particular, since one of the primary goals of the investigation is to demonstrate that there is no program-independent bound on the speedups achievable in production system programs, it is preferable to take the approach that makes it easy to extract the highest possible speedup. However, data specification languages remain a viable and interesting avenue for future work.

There is only one sequencing point in the execution of a production system program – the select phase, which selects the operation(s) to be performed in each cycle. It consists of two parts. The first part uses an ordering relation between instantiations to order the conflict set and the second part determines and extracts its *dominant subset*.⁴ Instantiations in the dominant subset are fired in the act phase.

Sequential languages use ordering relations that impose a total order on the conflict set, that is, cardinality of the dominant subset in sequential languages is 1. To allow an arbitrary number of instantiations to be executed per cycle, a parallel language must allow dominant subsets of unbounded cardinality. In other words, it must allow the programmer to specify ordering relations that define a partial order on the conflict set. For example, the partially ordered conflict set in Figure 3.5 has a dominant subset of cardinality five. Therefore, the design decisions to be made are (1) what kind of ordering relations should be permitted and (2) how they are to be specified.

The following subsections discuss the design alternatives.

⁴The dominant subset of D_S of a set S ordered by the relation \succeq is defined as $D_S = \{x \mid \forall y \in S, y \neq x \wedge y \not\succeq x\}$.

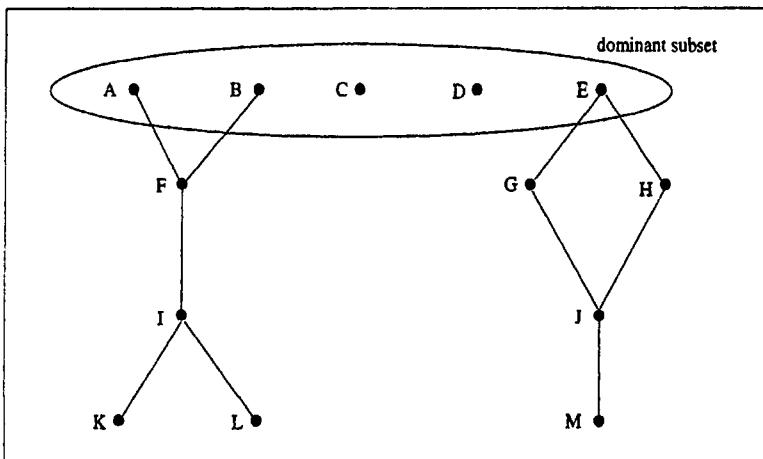


Figure 3.5: Partially ordered conflict set

3.2.1 Specification of ordering relations

There are two ways in which the ordering relation can be specified: declarative, that is, by explicit enumeration and procedural, that is by a procedure which given two instantiations either indicates the order between them or indicates that they are incomparable.

Explicit enumeration: The set of all instantiations is infinite. Therefore, it is impossible to explicitly enumerate the relationship between all pairs of instantiations. However, the set of productions in a program is finite. It is possible to explicitly enumerate a partial order relation from the set of productions in a program to itself and this order can be extended to the instantiations of these productions. One way of doing this would be to use a priority ordering which associates a priority with each production. The priority associated with a production is propagated to all its instantiations. Other possibilities include grouping productions and assigning priorities to groups, an explicit ordering sublanguage and so forth. An interesting approach to explicit enumeration has been taken by de Maindreville *et. al.* in the RDL/C system[15]. RDL/C provides a regular expression-like control language to specify ordering between productions. Given their static nature, explicit enumeration schemes are only able to specify the relationship between instantiations of different productions. It is unable to specify an order on multiple instantiations of the same production and they must be assumed to be incomparable. In other words, they must be assumed to not interfere with themselves (for

example of a simple production that interferes with itself, see Figure 3.1). Since it is rarely the case that all productions of a program are self-independent, this inflexibility renders explicit enumeration undesirable.

Procedural ordering: Procedural ordering delays ordering decisions until run-time when the instantiations are available. It is more powerful than explicit enumeration since it is able to order multiple instantiations of the same production. It can use a wide variety of properties of instantiations, ranging from their size and the timetags of the constituent tuples to values in the constituent tuples.

Procedural ordering can be supported in two ways – by providing a fixed ordering procedure as a part of the language or by providing a sublanguage to specify program-specific ordering procedures. Since a fixed ordering procedure has no knowledge about the productions, it can depend only on structural properties, like the size of instantiations, and on universal attributes like the timetags of the tuples. Therefore, it can be expected to order instantiations quickly, for reasonable ordering relations. Program-specific ordering procedures can be more discriminating than a fixed procedure by taking advantage of program-specific information. However, program-specific ordering procedures can be arbitrarily complex. This can significantly increase the time needed to order the instantiations as well as make such programs much more difficult to comprehend. In this author's opinion, the flexibility provided by program-specific procedures is not sufficient to offset its disadvantages. The PARULEL language [107] provides a production system sublanguage to specify the ordering procedure. In this author's opinion, most programs that have been written using PARULEL can be easily and more concisely written in a language based on a fixed ordering procedure. Hernandez and Stolfo present two PARULEL programs in [50]. Appendix E contains versions of these programs written in a language with a fixed ordering.

Specification of a partial ordering relation consists of two parts: specifying the ordering relation and specifying the domain of the ordering relation, that is, specifying which instantiations are comparable. The next subsection discusses the different ways of specifying the domain of the ordering relation.

3.2.2 Which instantiations are comparable

Intuitively, one might expect the domain of the ordering relation to be the set of all instantiations – that is, all instantiations can be compared with all other instantiations. This, however, requires the programmer to keep track of the partial order for the entire program. The data-dependent nature of the production system programs already makes it difficult for programmers to develop a good model of the flow of control in their programs. A real-life production system program can be expected to contain a few hundred to a few thousand productions. Keeping track of such large partial orders would greatly increase the complexity of programming and render such

languages unusable in practice. Furthermore, since ordering relations are transitive and since there are no limitations on the topology of the partial order, it is not possible to localize the effect of changes in the partial order. For example, consider the partial order in Figure 3.6. Suppose the programmer changes the program source so that instantiation A is no longer comparable with instantiation F. By transitivity of the partial order, A is no longer comparable with F, J, H and K, and can fire in parallel with them. This may or may not be intended. For unstructured partial orders, the programmer might have to check the partial order for the entire program before making a small change.

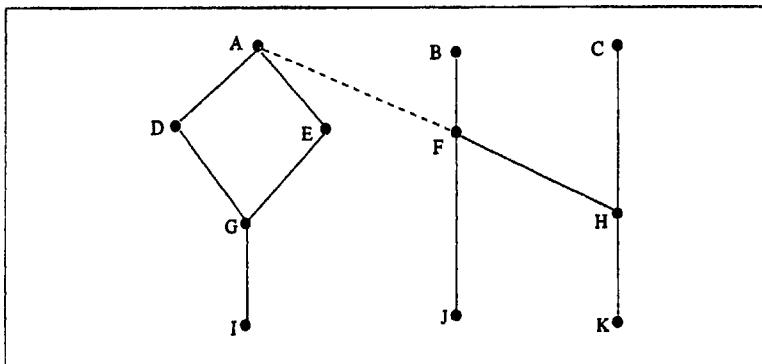


Figure 3.6: Effect of modifying an unstructured partial order

To keep the complexity of writing parallel production system programs under control, it is necessary to permit programmers to partition the partial order for the program into manageable chunks and to localize the effects of changes. This can be achieved by providing constructs that allow the programmer to partition the group of productions into independent subsets – all instantiations of productions from one set being incomparable with (or independent of) instantiations of productions from other sets. In the xor-gate simulator, each production can be encapsulated in its own partition indicating that xor-gates whose inputs are equal can be handled independently of xor-gates whose inputs are unequal.

Partitioning the programs into disjoint sets specifies only the relationship between instantiations of different productions. The relationship between instantiations of the same production can be specified at compile-time by annotating the productions whose instantiations are mutually independent; instantiations of unannotated productions are assumed to be mutually dependent. In the xor-gate simulator, both productions would be annotated since all xor-gates of either kind (equal inputs or unequal inputs) can be processed independently.

In conclusion, the desirable design choices for a parallel production system language are: *procedural ordering, fixed ordering procedure, partitioned partial orders* and *production annotation*. The next section describes the design of a language based on these design choices.

3.3 PPL

This section describes Parallel Production Language (PPL). The design of PPL instantiates the design choices for parallel constructs that were discussed in the previous section in the context of OPS5. OPS5 was selected as the base language for three reasons:

- It is important to study complete implementations of programs and not just kernels. OPS5 provides all the essential features of production system languages without any of the frills often associated with such languages. It is possible to build and study complete implementations of OPS5-like languages without worrying about the bells and whistles.
- To isolate the effect of parallelization on performance, it is important to share as much of the compiler and run-time system as possible between the sequential and parallel versions of the programs. All available test programs are written in OPS5. Since PPL is based on OPS5, they are also PPL programs (or almost so) and can share the PPL compiler and run-time system with the parallelized versions.
- Almost all research on efficient and parallel execution of production system programs has been based on OPS5 programs. Using OPS5 facilitates comparison with previous research.

Details on OPS5 can be found in Section 2.2.

The following subsection describes the PPL constructs and their use. The subsequent subsection discusses the expressiveness of these constructs. The final subsection compares them with parallel constructs in other languages.

3.3.1 Parallel constructs

PPL adds the following constructs to OPS5:

Parallel Productions: These productions are syntactically identical to OPS5 productions except that the initial keyword is *parp* instead of *p*. The new keyword indicates that all instantiations of these productions are independent and can be fired in parallel. Instantiations of productions defined using the familiar *p* keyword are fired in sequence, as in OPS5.

Production Sets: This is a simple grouping mechanism to partition the set of productions. Syntax for a production set is: { pset *production_set_id* *production_list* }. Instantiations of productions in a production set are ordered using the usual OPSS ordering procedure (see [11] or [29] for details of the OPSS ordering procedure). Instantiations of productions in different production sets are independent. In effect, the conflict set is partitioned along with the set of productions.

Figure 3.7 contains a PPL version of the xor-gate simulator. In this version, both the productions have been replaced by corresponding parallel productions and have been encapsulated in their own production sets. As a result, all instantiations of both productions can be fired in parallel and all xor-gates can be processed simultaneously.

```
(pset turn-xor-on
  (parp xor-gate-on
    (xor-gate ^in1 <input1> ^in2 <input2> ^out <output>)
    (line ^id <input1> ^value <v>)
    (line ^id <input2> ^value <v>)
    (line ^id <output>)
    ->
    (modify 4 ^value 0))
  )

(pset turn-xor-off
  (parp xor-gate-off
    (xor-gate ^in1 <input1> ^in2 <input2> ^out <output>)
    (line ^id <input1> ^value <v>)
    (line ^id <input2> ^value <> <v>)
    (line ^id <output>)
    ->
    (modify 4 ^value 1))
  )
```

Figure 3.7: PPL version of the xor-gate simulator

3.3.2 Expressiveness

Partitioning the program into disjoint sets of productions localizes the effect of changes to the partial order. However, since the instantiations within each production set are totally ordered, certain partial orders cannot occur (for example, the partial order in Figure 3.6). Since every partial order on the conflict set corresponds to a particular parallel execution, some parallel executions are not possible. This raises the question whether these constructs can express useful forms of parallelism. This subsection discusses some common forms of parallelism and shows how they can be expressed in PPL.

- **Data-parallelism:** Data-parallelism can be directly expressed by parallel productions. Multiple instantiations of a production match multiple data items, have different variable bindings but share a common set of actions. Firing them in parallel causes this common set of actions to be performed on all the matched data items. For example, see the PPL version of the xor-gate simulator in Figure 3.7. The data consists of the set of all xor-gates with two subsets of variable size: the collection of xor-gates whose inputs are equal and the collection of xor-gates whose inputs are unequal. The program in Figure 3.7 is able to process all the elements of this collection in parallel.
- **Pipeline-parallelism:** Pipeline-parallelism can be expressed by multiple production sets arranged such that productions in each production set match the tuples generated/processed by the productions in the previous production set. The program in Figure 3.8 illustrates how this can be done for a three-stage pipeline. Since the productions for each of the production sets are independent, up to three instantiations can fire per *msa* cycle, one for every stage. Each stage tags the objects processed by it and the productions for the subsequent stage match this tag to ensure sequencing of the operations in the pipeline.
- **Task-pile-parallelism:** Task-pile-parallelism can be expressed by a set of production sets — one production set for each task-type. A task is represented by a tuple that contains the necessary information and the tuple-space serves as the task-pile. Each task tuple matches a production in the production set that contains the code for the task. Firing the instantiation generates the processing for the task. If the productions in this production set are parallel productions, all similar tasks are processed in parallel. Strictly speaking, this is different from the standard task-pile model since the number of “workers” is not fixed and, conceptually, a new “worker” is created for every task. Figure 3.9 contains a PPL program that uses task-pile parallelism to compute Fibonacci numbers. The first production set implements the division step; it generates tasks to compute $\text{fib}(n-1)$ and $\text{fib}(n-2)$ given the task to compute $\text{fib}(n)$. The second production

```

{pset stage-1
  (p perform-stage-1
    (object ^stage-completed 0 ^data <d>)
    ->
    (modify 1 ^stage-completed 1 ^data (function-1 <d>)))
  }

{pset stage-2
  (p perform-stage-2
    (object ^stage-completed 1 ^data <d>)
    ->
    (modify 1 ^stage-completed 2 ^data (function-2 <d>)))
  }

{pset stage-3
  (p perform-stage-3
    (object ^stage-completed 2 ^data <d>)
    ->
    (modify 1 ^stage-completed 3 ^data (function-3 <d>)))
  }
}

```

Figure 3.8: Stripped down code for pipeline parallelism

set takes care of the base cases and third production set implements the combination step for fib(n) given the values for fib(n-1) and fib(n-2).⁵

3.3.3 Comparison with other parallel constructs

A parallel production scans the tuple-space for tuple combinations that satisfy the conjunction of conditions in its if-part. It performs the same set of operations on each such tuple combination it finds. In effect, a parallel production is a *comprehension* of the tuple-space, in the style of list and array comprehensions in functional languages. In general, a comprehension is an iterator which is characterized by a predicate and an operation. It scans aggregate data

⁵Of course, this is not the recommended way to compute Fibonacci numbers in PPL!

```

{pset create-tasks
  (parp division-step
    (fibonacci-number ^index <1> > 2) ^value -1 ^prev-index -1)
    ->
    (modify 1 ^prev-index (<1> -1))
    (make fibonacci-number ^index (<1> -1) ^value -1 ^prev-index (<1> -2))
    (make fibonacci-number ^index (<1> -2) ^value -1 ^prev-index -1))
  }

{pset execute-task-1
  (parp base-case
    (fibonacci-number ^index <= 2 ^value -1)
    ->
    (modify 1 ^value 1))
  }

{pset execute-task-2
  (parp combination-step
    (fibonacci-number ^index <1> ^value -1 ^prev-index <prev1>)
    (fibonacci-number ^index <prev1> ^value <v1> ^prev-index <prev2>)
    (fibonacci-number ^index <prev2> ^value <v2>)
    ->
    (modify 1 ^value (<v1> + <v2>)))
  }
}

```

Figure 3.9: Fibonacci using a task-pool approach

structures for items that satisfy the given predicate and performs the given operation for every such item that it finds. Comprehension is a parallel construct since each data item can be processed independently. Examples include the *list-comprehensions* of Miranda [120], list and array comprehensions of Haskell [52], *set-formers* of SETL [103] and the *apply-to-each* construct of NESL[8]. Figure 3.10 illustrates the analogy by presenting a NESL version of the xor-gate simulator.⁶ In this example, the two comprehensions in the function `simulate_xor` correspond to the two parallel productions in the PPL version.

⁶NESL provides `xor` as a boolean primitive. It is not used here to allow illustration of the analogy.

```

datatype xor_gate(int,int,int,int);
/* the fields are input1, input2, identifier and output */

function simulate_xor(xor_gate_seq) =
  {(x,x,id,0) : (x,y,id,_) in xor_gate_seq | x == y} ++
  {(x,y,id,1) : (x,y,id,_) in xor_gate_seq | x /= y}

/* ++ is the NESL operator for concatenation */

```

Figure 3.10: xor-gate simulator in NESL

Productions belonging to all production sets match and modify the same tuple space. However, each of them has its own conflict set, that is, its own locus of control. In effect, a production set is a separate *thread* and is suitable for expressing task level parallelism. Together, parallel productions and production sets support mixed (task and data) parallelism.

3.4 Implementation of PPL

This section describes `pplc`, an implementation of PPL. It is based on the parallel Rete algorithm described in Section 2.5. Section 3.4.1 describes the overall organization of the implementation. The primary goal of `pplc` was to minimize the fraction of time spent in sequential execution. The guiding principle in the design of the compiler and the run-time system was “pay only for what you use”. One of the consequences of this is that for a single processor, `pplc` reverts to an efficient uniprocessor implementation with no parallelization overheads.

3.4.1 Overall organization

`pplc` is a full implementation of PPL. It compiles PPL to portable C code. The run-time system is in portable C except for spin-lock routines which have to be rewritten for every processor. In addition to the constructs mentioned in the previous section, `pplc` also implements extensions for distributed memory execution for which it uses PVM[35]. It is largely independent of the operating system, the only dependency being a call to allocate memory regions shared between multiple processes. It runs on the Encore Multimax, multiprocessor Vaxen, Omron Lunas, uniprocessor Unix workstations and workstation clusters running PVM. It has been used to

compile programs ranging up to 50,000 lines in program length and several hours in (Alpha AXP) execution time.

`pplc` provides automatic parallelization of match and select phases. For sequential programs that do not use the parallel constructs provided by PPL, it is able to match, and often do better than, the speedup achieved by parallelizing implementations of sequential languages.

`pplc` overlaps all phases of a cycle between two barrier synchronizations. For matching, it uses the parallel Rete algorithm described in Section 2.5. It uses hashtable-based memory nodes and parallelizes match at the level of individual tokens. Most tokens are short, usually between 100-500 instructions. However, there is a significant variance. For conflict-set operations (i.e., the select phase), it uses a parallel algorithm based on two-level heaps (this is described in the next subsection). Individual tasks in this algorithm are usually substantially larger than match tasks (usually between 450 and 2500 instructions). The operations in the act phase are parallelized at a coarse-grain. All operations for a single firing constitute one task. This is required by the sequential semantics of the actions. Breaking down the act phase tasks is not desirable for languages based on OPSS as it leads to subtle race conditions. To be able to efficiently schedule the fine-grain tasks that are expected to dominate the processing, `pplc` does its own scheduling. It creates a process and a task-stack for every processor available to it.

`pplc` does its own memory management. It obtains memory from the operating systems in large chunks and maintains size-based free lists.

The following subsections describe the `pplc` compiler and run-time library.

3.4.2 The `pplc` compiler

The `pplc` compiler is modular and is organized as a sequence of walks over internal representations of the program. It uses two internal representations – an annotated parse tree and the Rete network. It currently has seven phases: parsing, source-to-source transformation, type inference, constraint propagation, generation of Rete network, optimization of Rete network and code generation. Additional phases (for analysis or optimization) can be easily inserted. The compiler can also be easily modified to handle similar languages.

The rest of this subsection describes some interesting phases of the compiler.

3.4.2.1 Source to source transformations

`pplc` performs two major source-level transformations – converting conditions to their canonical forms and constraint propagation.

Canonical conditions: Every field in a tuple, named or numeric, has a unique index. This transformation sorts the tests in a condition in increasing order of field indices and eliminates repetitions in the sequence of values for disjunctive tests.⁷ If there are multiple tests for a single field, they are merged into a single restriction list. If the resulting list has more than one disjunctive tests, they are merged. Figure 3.11 shows an example of this transformation. This transformation improves the performance in the following ways:

- It increases sharing in both the α and the β parts of the Rete network. Sharing the tests for common conditions (α tests) or condition prefixes (β tests) is one of the major advantages of Rete and its derivatives. Converting conditions to a canonical form makes it possible to find all conditions which are semantically identical, that is, perform the same set of tests. Without this transformation, only those conditions that are syntactically identical, that is, specify the same tests in the same order, can be shared.
- It increases temporal locality since all tests on a single field are performed together.
- It increases spatial locality since the tests on multiple fields of a single tuple are performed in increasing order of field index.

Constraint propagation: Every test in a production can be taken to specify a constraint on a field, the scope of the constraint being the production in which the test occurs. If there are multiple tests on the same field (specified in different conditions), it is possible to simplify the tests by collecting and propagating the constraints. This can be achieved in the following ways:

- Replacing variable tests by constants. This can be done if one of the occurrences of the variable is tested for equality with a constant or a group of constants. Figure 3.12 shows an example for the variable $\langle v1 \rangle$. Variable tests are implemented by β nodes and are far more expensive than constant tests which are implemented by α nodes. The difference of cost between the two cases can often be as large as two orders of magnitude.
- Increase selectivity of tests occurring early in the production. This can be done by if a later occurrence of a variable has a more restrictive test than earlier ones. Figure 3.12 shows an example for the variable $\langle v2 \rangle$. Increasing the selectivity of an early test reduces the number of partial matches generated. This reduces the stress on the state-maintenance algorithms.

It is also possible to detect, at compile-time, if the conditions in a production are inconsistent. Such productions can never be matched. Figure 3.13 shows an example. The test in the second

⁷Disjunctive tests are specified by a sequence of values bracketed by << and >>.

```
Field indices: (tuple-type-1 field1 = 1 field2 = 2 field3 = 3)
              (tuple-type-2 field4 = 1 field5 = 2)

(p non-canonical
  (tuple-type-1 ^field3 <v> ^field2 > 26 ^field3 < 36)
  (tuple-type-2 ^field4 <<x a b>> ^field5 <v> ^field4 <b c>>)
  -->
  (some actions))

(p canonical
  (tuple-type-1 ^field2 > 26 ^field3 {<v> < 36})
  (tuple-type-2 ^field4 <<x a b c>> ^field5 <v>)
  -->
  (some actions))
```

Figure 3.11: Example of the *canonical conditions* transformation

```
(p before-propagation
  (tuple-type-1 ^field1 <v1> ^field2 <v2>)
  (tuple-type-2 ^field4 {<v1> <<a b>>} ^field5 {<v2> < 36})
  -->
  (some actions))

(p after-propagation
  (tuple-type-1 ^field1 <<a b>> ^field2 {<v2> < 36})
  (tuple-type-2 ^field4 <<a b>> ^field5 <v2>)
  -->
  (some actions))
```

Figure 3.12: Example of constraint propagation

condition specifies that the value bound to $\langle v2 \rangle$ be less than the value bound to $\langle v1 \rangle$. The test in the third condition specifies that the value bound to $\langle v3 \rangle$ be less than the value bound to $\langle v2 \rangle$ and greater than the value bound to $\langle v1 \rangle$. Since any value less than $\langle v2 \rangle$ is also less than $\langle v1 \rangle$, the third condition can not be matched.

```
(p inconsistent
  (tuple-type-1 ^field1 <v1>)
  (tuple-type-2 ^field4 (<v2> < <v1>))
  (tuple-type-3 ^field6 (<v3> < <v2> > <v1>))
  -->
  (some actions))
```

Figure 3.13: Example of a production with inconsistent conditions

3.4.2.2 Type inference

At the lowest implementation level, the match procedure consists of binary tests, equality or relational, between fields of tuples. In the absence of information about the types of the fields involved, every comparison has to be preceded by a type tag check which ensures that the values being compared are of the same type. The goal of type inference is to eliminate these checks by extracting type information from the program.

PPL tuples can contain only integer, symbolic or floating point values and can not be nested. Therefore, the type lattice for PPL programs is simple. It contains these three types bracketed by a top and bottom element. `pplc` infers the types for the fields of tuples, the variables occurring in the program and the foreign functions called. `pplc` uses the following axioms for type inference:

- A field can contain values of only one type.
- A variable can be bound to values of only one type.
- A field tested against a constant literal must be of the same type as the literal.
- If a relational test is applied to a field, it cannot contain symbolic values.
- If a field is tested against more than one variable, all such variables must have the same type.

- Functions called from PPL programs are first-order and monomorphic.

The sources of information for type inference include constant literals in both the if-parts and the then-parts of the productions, calls to primitive functions (whose types are known) and type declarations, if any. Type information is iteratively propagated till no new inference can be made. Type inference converges rapidly (within three iterations) for all programs that it has been tested on. Figure 3.14 shows an example of the type inference in `pplc`. Since the variable `<v>` appears in field1 of tuple-type-1 (first condition), field5 of tuple-type-2 (second condition) and field4 of tuple-type-2 (the make action), these fields are of the same type. In second condition, field4 is tested against a symbolic constant. Therefore, all these fields must be symbolic. Since field2 and field3 of tuple-type-1 are tested against integer constants, they must both be integral. Now the types for all fields are known.

```
(p type-inference-example
  (tuple-type-1 ^field1 <v> ^field2 72 ^field3 <2>
  (tuple-type-2 ^field4 abc ^field5 <v>)
  -->
  (make tuple-type-2 ^field4 <v>))

Result of type inference:
  (tuple-type-1 field1 : symbol field2 : integer field3 : integer)
  (tuple-type-2 field4 : symbol field5 : symbol)
```

Figure 3.14: Example of type inference

3.4.2.3 Code generation

`pplc` generates C code for all the tests specified by the Rete network and for all the actions in the then-parts of the productions. Code for the Rete network is inlined; separate code is generated for every α and β node. Code for the right hand side actions is not inlined and consists mainly of calls to run-time library routines.

The code generation phase implements the following optimizations:

- Frequently accessed values are cached in local variables. Examples of this include the value(s) being tested during the traversal of a hash bucket, the value being tested against a

set of constants (in a disjunctive test) and pointers to tuples which operated on by several actions.

- For α networks with large branching factors, hashing is used to reduce the number of conditions that are tested against each tuple. The hash function uses the constant literals occurring in the conditions to partition the set of conditions. Figure 3.15 shows an example. In this case, instead of being tested against all four conditions, each tuple is tested against only one condition.

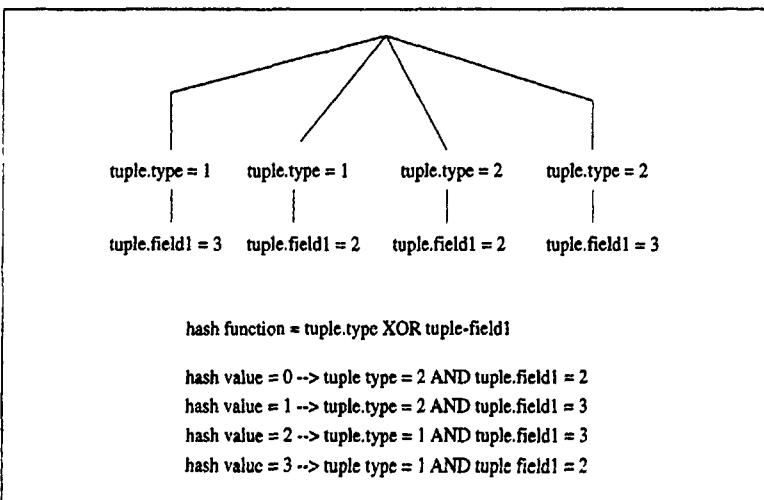


Figure 3.15: Example of hashed α network

3.4.3 The pplc run-time library

As the size of the data set grows, the parts of the implementation that are stressed the most are the algorithms to maintain the match state, that is, algorithms for managing the memory nodes and the conflict set. Section 3.4.4 describes the state-maintenance algorithms used in the pplc run-time library.

In addition to using parallel state-maintenance algorithms, a scalable parallel implementation also has to minimize the time spent in resource contention. Section 3.4.5 describes how the `pplc` run-time library attempts to minimize resource contention.

3.4.4 Parallel state-maintenance algorithms

`pplc` stores the match state in five parts: memory nodes to hold tokens, conjugate memory nodes to hold out-of-order token deletion requests, conflict set to hold the active instantiations,⁸ *refracted* conflict set to hold the inactive instantiations and *conjugate* conflict set to hold the out-of-order instantiation deletion requests.

Memory nodes: The `pplc` run-time library uses two global hashtables to store the contents of the memory nodes. A unique identifier is assigned to every β node in the Rete network. Identifier of the destination β node and the values of the fields being tested at the β node are used as parameters to the hash function. Gupta *et al.* claim that storing the contents of the memory nodes in hashtables can reduce the number of comparisons required to search a memory node for a matching token by up to ten fold [42]. The `pplc` run-time library uses large hashables (64K buckets) to reduce the probability of collisions. Even with large hashables, good distribution of tokens to hash buckets is critical for good performance. The hash function used xors the arguments together with a large prime number to ensure a good distribution.

Conjugate memory nodes: The `pplc` run-time library uses two global hashables to store the contents of the conjugate memory nodes. Since the only operation that is performed on out-of-order token deletion requests is to match them, hashables perform fairly well. Conjugate hashables mirror the main hashables and share the hash function which allows the cost of computing the hash function to amortized over accesses to all four hashables.

Conflict set: The `pplc` run-time library uses a separate conflict set for every production set. For individual conflict sets, it uses a two-level heap. Corresponding to every production in the production set, there is a heap for its instantiations. The conflict set itself is a heap of such heaps. Figure 3.16 shows a sample conflict set for a production set with five active productions. In practice, the upper level heap consists of the top instantiations for each active production. The worst-case performance of a two-level heap is identical to that of a conventional heap. The actual performance depends on the distribution of the instantiations and the ordering relation used. Locking is done separately for each production-heap and the top-level heap.

Refracted conflict set: The refracted conflict set contains the instantiations that have already been fired. Instantiations are added to the refracted conflict set after they have been fired and they are deleted from this set when an identical instantiation with a *deletion* tag arrives at the conflict set (this happens when a tuple matching a non-negated condition is deleted or a

⁸Instantiations that have not yet been fired.

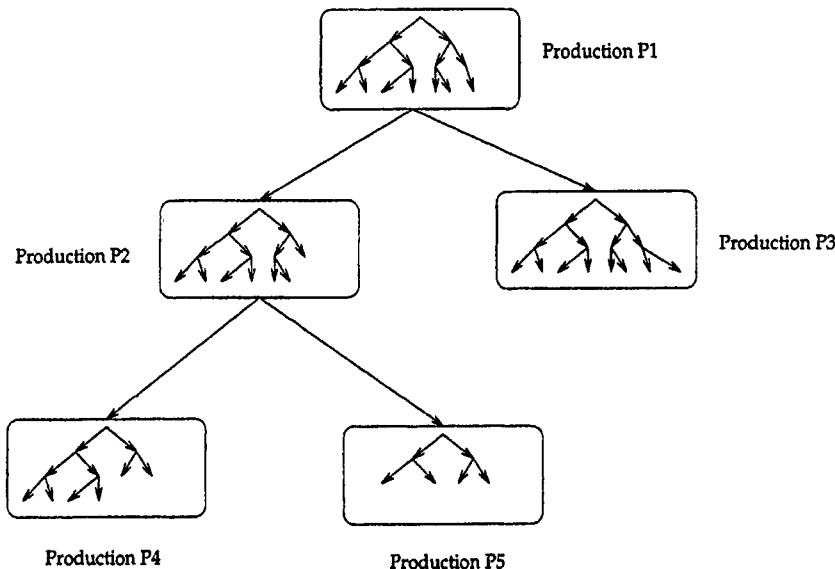


Figure 3.16: Example of a two-level heap-based conflict set

tuple matching a negated condition is added). The `pplc` run-time library uses one hashtable per production to implement the refracted conflict set. The hash function xors the timetags of the constituent tuples. A null timetag is used as a placeholder for negated conditions. Hashtables are suitable for the refracted conflict set since the only operations on it are addition and searching for the purpose of deletion.

Conjugate conflict set: Like the refracted conflict set, the only operations on conjugate conflict sets (which store out-of-order deletion requests for instantiations) are addition and searching for the purpose of deletion. Consequently, the `pplc` run-time library implements them in a similar manner.

3.4.5 Minimizing resource contention

There are seven shared resources: the memory nodes (including the conjugate memory nodes), conflict sets (including the active, refracted and conjugate conflict sets), the symbol table, free

memory, the task stacks, the tuple-space and the counter for generating the unique timetags for the tuples. This section describes the efforts that have been made to reduce contention for each of these resources.

Memory nodes: Four global hash-tables are used to implement memory nodes – two to hold the contents of the left and right memory nodes and two to hold the contents of the corresponding conjugate memory nodes. All four hash-tables share a common hash function. Therefore, processing a single token requires the contents of only one *line* of the hash-tables – a line being the set of buckets with the same index from all the four hash-tables. A single simple lock protects each line. Even though it is possible to overlap multiple read accesses using *multiple-reader-single-writer* locks, the `pplc` run-time library uses simple locks since the complex locks have been shown to increase the execution time for OPSS programs which have high available parallelism [42]. Similar results have been shown for parallel implementations of Soar programs [113].

Conflict sets: Each heap in the conflict set is separately locked. This avoids both monolithic locking, which serializes all updates to the conflict set, and instantiation-level locking which would cause frequent locking and unlocking. Since locking requires exclusive access to the bus (or other shared communication medium), frequent locking can greatly reduce the bandwidth for the locking process as well as for other processes.

Symbol table: since almost all accesses to the symbol table are read-only, there is almost no contention for the symbol table. The `pplc` run-time library uses a separate counter and a unique prefix for every process to generate new uninterned symbols. Therefore, there is no contention for *gensyming* new symbolic constants.

Free memory: The `pplc` run-time library manages its memory on a per-process basis. Each process has its own chunk of memory and size-specific free lists. When any process runs out of memory, it independently obtains another chunk from the operating system. Hence, there is no contention for free memory.

Task-stacks: The `pplc` run-time library uses a separate stack for every process. Every process adds tasks only to its own task-stack and as far as possible, removes tasks only from its own task-stack. When its task-stack is empty, a process scans the rest of the task-stacks in a round-robin fashion. For programs that have a large number of tasks, little contention can be expected for task-stacks. Gupta and his colleagues present similar results for a parallelizing implementation of OPSS in [42] and for a parallel implementation of Soar in [113].

Tuple-space: The tuple-space is accessed during the match and act phases. Accesses during the match phase are exclusively read accesses; accesses during the act phase are mixed. Since the production system semantics require that all instantiations being fired in a cycle see exactly the same view of the tuple-space (in other words, all instantiations fire together), deleted tuples are not freed till the next barrier synchronization which occurs at the end of the subsequent match phase. Since all read accesses are directly through a pointer to the tuple, there is no

need to lock the tuple-space for read accesses. Since multiple deletions of the same tuple in a single *msa* cycle are legal, deletions too are permitted without locking. Additions to the tuple-space, however, require locking. The tuple-space is implemented as a list of tuples and adding a tuple involves consing it onto this list. Since consing is cheap, and since read accesses overwhelmingly dominate write accesses, the contention for the tuple-space lock is expected to be low.

Timetag assignment: By its very nature, the assignment of timetags is serial. To assign a new timetag, it is necessary to acquire a lock, read and increment a counter and release the lock. Since, incrementing the counter is a cheap operation, it is expected that the contention for the counter will not affect the execution greatly. The effect of this serialization can be further mitigated by the use of an atomic Fetch-and-Add instruction for architectures on which it is available (e.g. the Ultracomputer [32]).

Chapter 4

Parallelism Experiments

The goal of these experiments was to test three hypotheses. First, that there is no program-independent bound on the parallelism available in production system programs. Like in other paradigms, the parallelism available in production system programs depends on the parallelism inherent in the program and the way the program has been encoded. To help verify this hypothesis, a diverse set of programs, including embarrassingly parallel programs as well as programs with large non-parallelizable loops, was included in the benchmark suite.

Second, that the parallelism available in a production system program can scale with data. That is, parallelism is a possible solution for the problem of dealing with large data sets. To help verify this hypothesis, programs that process scalable data sets were included in the benchmark suite.

Third, that production sets and parallel productions are effective for the expression of parallelism in production system programs.

This chapter describes the benchmark suite and the structure of the experiments. The next chapter presents and analyzes the results. Section 4.1 describes the benchmark suite. It describes the programs, the parallelization strategy and the data sets used in the experiments. Section 4.2 describes the structure of the experiments. Section 4.3 describes the simulator used in these experiments to simulate the execution of PPL programs on a multiprocessor.

4.1 Benchmark suite

The benchmark suite has two classes of programs: programs with scalable data sets, that is data sets which can be characterized by a numerical parameter and which can be scaled by assigning increasing values to the parameter, and programs with data sets which cannot be thus characterized but are large. The benchmark suite has four programs that process parameterized

data sets and one that processes non-parameterized data sets. To ensure an efficient baseline, the sequential versions of all the programs included in the suite have been substantially optimized before being used in the experiments. The following subsections describe the programs in the benchmark suite in some detail. Code for the first four benchmarks can be found in Appendix E.

4.1.1 Circuit simulator (`circuit`)

This program simulates a gate-level circuit with a constant-delay model. Currently, it simulates circuits with two-input gates but it can easily be extended to any desired logic device. It does not support the *implied-or* logic provided by open-collector TTL devices. Execution of `circuit` has two phases – the first in which the operation of all devices is simulated and the second in which the values generated by the first phase are propagated down the lines. This program is an optimized version; the original version was written by Dan Neumann of the University of Massachusetts, Amherst. The primary optimizations are:

- Modify only those tuples that correspond to lines whose value has to be changed. The original version modified tuples corresponding to all lines. This optimization reduced the amount of parallelism available as fewer tuples are being modified.
- Replace variable tests by constant tests by creating several copies of productions. This is possible for `circuit` since there are only a small number of cases for each device type. This moves testing for input cases from the β network to the α network. Since all α tests are performed as a single task, this also reduces the number of schedulable tasks.

These optimizations improved the performance of the program (on a Decstation 5000/200) by 1.87 fold for the smallest data set and by 2.25 fold for the largest data set.

Parallelization: Since each device and each line can be simulated independently, the productions for each device class and for the interconnection lines were encapsulated in separate production sets and were converted to parallel productions. Simulation of the circuit in each cycle has to be atomic. All the values in a given simulation cycle must be based only on the values from the previous cycle. To ensure this, the sequential version of the program does not delete the tuples corresponding to the old values while it is computing the new values. Instead, it creates copies of the tuples corresponding to the lines whose value has changed and modifies these copies. The old values are deleted at the end of the first phase. Since the parallel version is able to update collections of tuples in a single α cycle, it does not create these copies. Therefore, each phase of a simulation cycle is performed in a single msa cycle. The information that is needed to successfully parallelize `circuit` is the fact that all lines have only one driver and therefore can be safely updated simultaneously.

Data set: For these experiments, the simulator was run on 200 cycles of a linear feedback shift register with random initial state. The data set parameter was the size of the linear feedback

shift register. Figure 4.1 shows the linear feedback register of size three. The linear feedback shift register is a pseudo-random number generator. As a result, the values of the lines change in an irregular manner. Since the number of modifications governs the amount of work done and the parallelism available, this benchmark can be expected to show irregular parallelism. The largest register simulated had 275 cells.

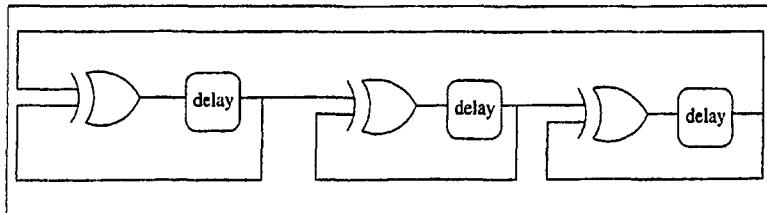


Figure 4.1: Linear feedback shift register of size three

4.1.2 Game of Life (life)

This program implements Conway's "Game of Life" which computes the state of a matrix of simple cellular automata. Each matrix cell can either be dead or alive. The future state of a cell depends on its own state and the state of its immediate rectilinear neighbors (neighbors along the NEWS directions) and is given by the following rules:

1. If a live cell has less than two live neighbors in any cycle, it dies of loneliness.
2. If a live cell has two or three live neighbors in any cycle, it remains live.
3. If a dead cell has three live neighbors, it becomes live.
4. If a live cell has four neighbors, it dies of overcrowdedness.

Execution of `life` consists of two phases, the operational phase followed by the print phase. The operational phase consists of a sequence of operation cycles which compute the state of cellular automata. In each operation cycle, the state of the entire matrix is atomically updated. The print phase prints the state of the entire matrix. This program is a substantially rewritten and optimized version; the original program is available from the Department of Computer Science, Columbia University (`ftp.cs.columbia.edu:pub/prosys/prosys.tar.Z`). The primary optimizations are:

- The original program computed the number of live neighbors for each cell in each operation cycle. This is unnecessary as there are only four neighbors for a cell and this information can be encoded in the productions. This optimization reduced the amount of parallelism available as parallel computation of neighbors is no longer being done.
- Modify only those tuples that correspond to cells whose status has changed. The original version modified tuples corresponding to all cells. This optimization also reduced the amount of parallelism available as fewer tuples are being modified.
- Replace variable tests by constant tests by creating several copies of productions. This is possible for `life` since there are only two cases for the status of each cells. This moves testing for input cases from the β network to the α network. Since all α tests are performed as a single task, this also reduces the number of schedulable tasks.

These optimizations improved the performance of `life` (on a Decstation 5000/200) by 17.23 fold for the smallest data set and 18.49 fold for the largest data set .

Parallelization: Since the next state of each cell can be computed independently and since the rules for computing the next state handle disjoint cases, all productions can fire in parallel. Therefore, all productions that implement the transition function are converted to parallel productions and are encapsulated in separate production sets. State transition of the cellular automata in each operation cycle has to be atomic. All the values in a given operation cycle must be based only on the values from the previous cycle. To ensure this, the sequential version of the program does not delete the tuples corresponding to the old values while it is computing the new values. Instead, it creates copies of the tuples corresponding to the lines whose value has changed and modifies these copies. The old values are deleted at the end of the first phase. Since the parallel version is able to update collections of tuples in a single `msa` cycle, it does not create these copies. Therefore, each operation cycle is performed in a single `msa` cycle. The information that is needed to successfully parallelize `life` is the fact that all cells have only one neighbor in each direction. This implies that at a time only one rule is applicable to a cell. Only the operational phase of the program has been parallelized. It is not possible to parallelize the printing phase.

Data set: For these experiments, `life` was run on 200 operational cycles on fixed size matrices with an initial state that leads to oscillations and a border of dead cells. The basic data set was the simple oscillating pattern in Figure 4.2 where, a “.” stands for a dead cell and a “*” stands for a live one. The data set parameter was the number of repetitions of this basic pattern. Since the oscillating cells change state every operation cycle and since no other cells change state, this benchmark can be expected to show regular parallelism. The largest matrix used had 70 repetitions of the basic pattern.

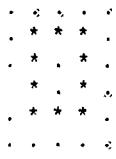


Figure 4.2: The basic pattern for the life data set.

4.1.3 Waltz labeling (waltz)

This program implements the Waltz labeling algorithm for interpreting line drawings [124]. This algorithm uses constraint propagation to eliminate a large number of possibilities. The input to waltz consists of junctions and lines between them, the output is a labeling for the junctions and lines which uniquely determines the orientation of the planes in the drawing. Execution of waltz consists of a sequence of constraint propagation phases. In each phase, it labels a set of junctions and propagates the constraints generated by this labeling to all lines incident on the junctions. At the end of the computation, it prints out the labeling. The original version of this program was written by Toru Ishida at Columbia University. It has since been revised by Dan Neumann of the University of Massachusetts, Amherst. The program used as a benchmark is an optimized version of the revision. The primary optimization was to reorder the conditions for most of the productions. This reduced the number of tokens being generated, i.e. the number of schedulable tasks. This optimization improved the performance of waltz (on a Decstation 5000/200) by 1.19 fold for the smallest data set and 1.89 fold for the largest data set .

Parallelization: Like the sequential version, the parallel version assigns a label to one of the boundary junctions and propagates constraints from there. Unlike the sequential version, the parallel version propagates constraints simultaneously along all lines incident on a junction. However, there is interference between the different constraints generated by a single junction labeling. Therefore, all constraints of a particular type can be propagated in parallel but constraints of different types have to be applied in sequence to achieve the desired result. To implement this, all productions that implement constraints are all converted to parallel productions but they are not encapsulated in separate production sets. Therefore, all instances of a single type of constraint can be propagated in parallel.

Data set: The line drawings used in these experiments were generated by repeating a basic block which is shown in Figure 4.3. The data set parameter was the number of repetitions of this basic block. The largest drawing used had 120 repetitions of the basic block.

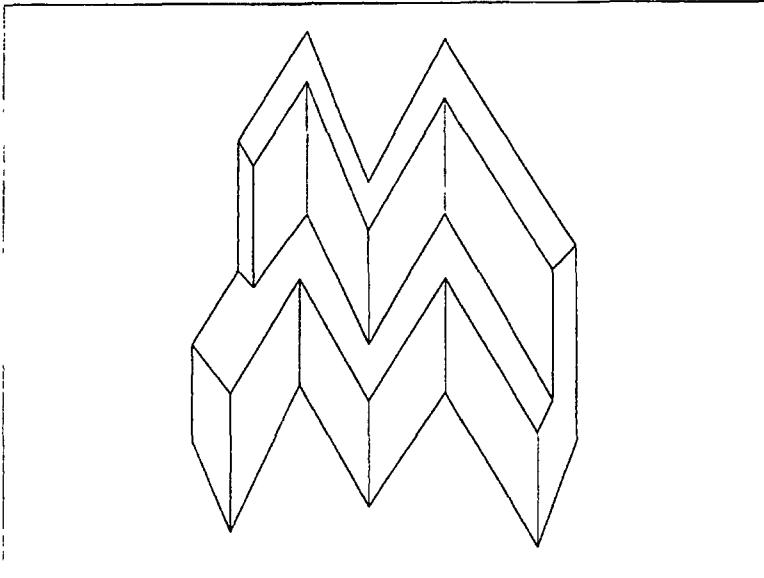


Figure 4.3: The basic block for the *waltz* data set.

4.1.4 Simulation of a hotel (hotel)

This program simulates the operations of a large hotel for one day – reservations, checkin, maid service, laundry, banquet etc. Execution of *hotel* consists of a sequence of phases corresponding to the phases in the operation of the hotel. This program is an optimized version; the original was written by Steve Kuo of the University of Southern California. The primary optimizations are:

- Source code for the original version had been blown up to over ten times its actual size by repeated applications of the *copy-and-constrain* optimization [92]. *Copy-and-constrain* is the production system analogue of inline expansion of function calls. In the optimized version, the inline expansion has been eliminated.
- Most of the information being processed by *hotel* is stored in *room* tuples. Since different sets of productions processed different parts of the tuple, modifying one part of

the tuple causes needless attempts to match the new tuple against productions that modify other parts of the tuple. The optimized version splits the room tuples into smaller pieces, each of which is separately processed.

- Processing for several operations was being performed piecemeal. This required needless generation and matching of intermediate tuples. These operations were consolidated.

These optimizations improved the performance of `hotel` (on a Decstation 5000/200) by 1.22 fold for the smallest data set and 8.41 fold for the largest data set.

Parallelization: `hotel` is significantly more complex than the programs previously discussed. Its phases were individually parallelized based on the semantics of the simulation – what operations can be performed in parallel in a hotel. There are several sequential loops – for example, collecting and doing the laundry, setting the tables for the banquet, cleaning restrooms. Some of these update counters, for example collecting and doing the laundry.

Data set: The data for `hotel` consists of several lists containing information about the rooms of the hotel, guests checking in, guests checking out, guests staying over, banquets scheduled, menus for the banquets etc. For these experiments, the ratios between the cardinalities of these lists have been fixed (the hotel is assumed to have 66% occupancy, each floor has 100 rooms, a fixed percentage of them will check out etc.). The data set parameter is the number of floors. The largest value of the parameter used was 10 floors.

4.1.5 Interpretation of aerial images (`spam`)

This program interprets aerial images of airports. The input to `spam` is a list of image regions, their positions and properties and the output is a model of the airport which can then be compared with known models to identify the airport. It builds the airport model by bottom-up pruning of possible interpretations for individual image regions. For pruning, it uses geometric constraints. It uses productions to determine the applicable constraints.

`spam` has four phases. The first phase applies per-region constraints to generate possible interpretations. There are up to 15 interpretations possible for every region. The second phase applies pair-wise constraints to prune some of the interpretations. There are 33 such constraints. The third phase clusters regions into functional areas, like runways with their associated taxiways and grassy areas, and merges overlapping functional areas. The final phase combines the functional areas to generate a model for the airport. `spam` has been developed by the MAPS group in the School of Computer Science, Carnegie Mellon University [75]. Due to its computationally intensive nature, `spam` has been a focus of parallelism research for the last six years. Harvey *et al.* [48] report on the parallelization of the first two phases of an OPSS version on the Encore Multimax. This effort achieved up to 13 fold speedup on 14 processors.

Currently, an implementation of *spam* on the *Midway* distributed shared memory system [6] is underway.

For this benchmark, the first three phases were used. The final phase is currently in a state of flux [47]. Furthermore, it is mostly in C which limits its utility as a production system benchmark. The first three phases dominate the computation, taking usually up to 90% of the time. The program used as the benchmark is an optimized version of the original program obtained from the MAPS group. Most of the optimizations were enabled by the parallel semantics of PPL which allow atomic update of aggregates in a single *msa* cycle. Other optimizations include condition reordering, common subexpression removal, replacement of function calls whose value is known at compile-time by their results, and optimizations to the foreign functions that implement the geometric constraints. These optimizations improved the performance of *spam* (on a Decstation 5000/200) by 1.55 fold for the smallest data set and 2.19 fold for the largest data set. This speedup is in spite of the extremely large cross-products generated in the first and second phase of the new version. Corresponding phases of the old version avoided these cross-products by processing the data piecemeal.

Parallelization: The first phase is a triply nested loop. For every region and every possible interpretation (fifteen of them), it checks if the values of twelve features are within a given range. Feature checks for all the interpretations are implemented by individual productions that match tuples corresponding to individual regions and call foreign functions to perform the checks for the matched region. If all the twelve checks succeed, a hypothesis tuple is generated to represent the particular interpretation of the region. Since no data for any region is modified, all levels of this loop can be parallelized. This is achieved by converting every feature-check production to a parallel production and encapsulating each of them in its own production set. This allows all feature-checks to be done in parallel. Figure 4.4 shows one such production that checks the area of the region for a runway interpretation. Calls to *spam_rtf_match_feature()* check if the value of the feature is within the range [*<1bound>*, *<ubound>*] and accumulate the result. For each region, this routine is called twelve times, once for every feature-check. These calls are not functional – they update an accumulator. To allow these calls to proceed in parallel, updates to the accumulator are protected by a lock. This is within the C code for the *spam_rtf_match_feature()* procedure.

The second phase applies constraints between pairs of hypotheses. Only some combinations of hypotheses are checked. For example, if a region has been hypothesized to be a runway, then all regions that have been hypothesized to be taxiways and that lie within a given distance from it should be perpendicular to it. There are thirty-three such constraints. Each of these constraints are applicable only to particular pairs of hypothesis types. The constraint mentioned above is applicable only to runways (the object type) and taxiways (the target type). This phase is also a triply-nested loop. For every hypothesis, for every applicable constraint, for every hypothesis of the target type, it applies the constraint on the object and the target hypotheses. This is implemented by a pair of productions for every constraint, one to apply the constraint

```

{pset runway-test-area
  (parp RTF**runway-match-area
    (region ^name <name> ^area <area> ^identifier <id>
      (rtf-rule-constants ^ruleset runway-match-attnbutes
        ^attribute area ^constants <lbound> <ubound>
      (rtf-stage ^name match-features)
      -->
      (call spam_rtf_match_feature <id> <area> <lbound> <ubound> runway 0))
    )
}

```

Figure 4.4: Sample production from first phase of SPAM

and one to check if constraint was satisfied. Since the data for the hypotheses is not changed in this process, all constraint applications can be done in parallel. This is achieved by converting each of the productions to parallel productions and encapsulating them in separate production sets. Figure 4.5 shows one such production that checks if a region hypothesized to be a taxiway is perpendicular to another region hypothesized to be a runway and is within 10000 units of distance from it. Since all constraints in this phase are purely functional, parallel calls can be safely made to the routine that applies the constraints, `spam.lcc.do.geometric.test()`.

The third phase attempts to build hypotheses for functional areas based on the hypotheses for individual regions. Each functional area has a *seed* region – runway, road, terminal-building and hangar-building. It has several loops: to select seed regions, to generate links between the seeds and the surrounding regions, to merge multiple parallel links between regions, to create functional areas and evaluate them, to determine overlapping areas and merge them. All of these loops except the loops that merge the links and the functional areas can be parallelized. The foreign function calls in this phase perform geometric operations like computing convex hull of functional areas and determining the degree of overlap between two regions. All of them are purely functional and multiple calls to them are safe.

Data set: Data from three images, corresponding to the Moffett airforce base (`moffett1`), the Washington National airport (`dc36809`) and the San Francisco International airport (`sf4917`), were used in these experiments. The first two are about the same size, `dc36809` being slightly larger. The third, `sf4917`, is substantially larger. This data was obtained from the MAPS group and had been hand-generated from the respective images.

```
{pset runways-orthogonal-taxiways
  (parp LCC-runways-are-orthogonal-to-taxiways
    (lcc-stage ^name apply-constraint)
    (lcc-rule-constants ^rulename runways-are-orthogonal-to-taxiways
      ^min <min> ^max <max> ^bound <bound>)
    (fragment ^hypothesis runway ^identifier <id0>)
    (fragment ^hypothesis taxiway ^identifier {<id1> <> <id0>})
    ->
    (make lcc-match-score ^rulename runways-are-orthogonal-to-taxiways
      ^result (spam_lcc_do_geometric_test 12 <min> <max> <bound> 10000)
      ^from <id0> ^to <id1>)
  )
}
```

Figure 4.5: Sample production from second phase of SPAM

4.2 Design of the experiments

These experiments measure and compare the speedups achieved by automatic parallelization and explicit specification. To freely vary the number of processors, these experiments used a trace-driven multiprocessor simulator for measuring the speedups. Simulation also allowed measurement code to be freely added without worrying about the distortion introduced. The only actual multiprocessor available for this investigation was an ancient 8-processor Vax which was in a fairly unstable condition.

For each benchmark, four versions were used:

1. **Sequential program running on a uniprocessor:** this version is generated by compiling the sequential version of the program using the PPL compiler targeted towards uniprocessors. It is used as the baseline for computing all speedups and is referred to as the **baseline** version.
2. **Sequential program running on a multiprocessor:** this version is generated by compiling the sequential version of the program using the PPL compiler targeted towards multiprocessors. It is used to compute the speedups achieved by automatic parallelization and is referred to as the **parallel-match** version.
3. **Parallel program running on a uniprocessor:** this version is generated by compiling the parallel version of the program using the PPL compiler targeted towards uniprocessors. It

is used to measure the effect of the parallel programming style on uniprocessor execution. It is referred to as the `parallel`-model version.

4. **Parallel program running on a multiprocessor:** this version is generated by compiling the parallel version of the program using the PPL compiler targeted towards multiprocessors. It is used to measure the speedups achieved by explicit specification of parallelism and is referred to as the `parallel` version.

All versions were compiled at the highest level of optimization of the PPL compiler. The C code generated by the compiler as well as the code for the run-time library was compiled using `gcc -O`. These settings are same as the ones used for generating the cost model.

Comparative benchmarking is prone to pitfalls. The most common pitfall is an inefficient baseline. To ensure meaningful results, the `baseline` version was used to compute all speedups. To the best of my knowledge, the uniprocessor implementation of PPL is faster than any other publicly available implementation of OPSS. Tables 4.1 and 4.2 compare the uniprocessor implementation of PPL with CParaOPSS, the C-based implementation of OPSS available from Carnegie Mellon University (`dravido.soor.cs.cmu.edu:/usr/nemo/cparaops5`). Table 4.1 contains results for three programs that process fixed data sets. `rubik`, written by James Allen solves the Rubik's cube problem for a particular configuration, `tourney`, written by Bill Barabash, schedules a bridge tournament for 16 players and `gen_tsp`, written by Jose Nelson Amaral, finds a travelling salesman tour of a collection of cities in the four southern US states. These programs have been used as benchmarks by production system researchers. Table 4.1 shows that for these benchmarks, the uniprocessor implementation of PPL is between 1.6 and 4 times faster than CParaOPSS and uses between 2.25 and 4 times less space. Table 4.2 contains results for three programs that process variable sized data sets. These programs have been used as benchmarks in a different part of this dissertation (see Chapter 8 for details). The first program, `make-teams`, operates on a database of employees and creates teams given some constraints on their composition. The numbers shown in the table correspond to a database of eighty employees. The second program, `clusters`, operates on a collection of image regions and groups them into clusters based on their distance from a group of seed regions. The numbers shown in the table correspond to an image with 400 regions. The third program, `airline-route`, operates on a airline flight database and determines the best available flight for a single traveller. The numbers shown in the table correspond to a database of 150 flights between 20 airports. All these programs were written by Milind Tambe. Table 4.2 shows that for these benchmarks, the uniprocessor implementation of PPL is between 2 and 90 times faster than CParaOPSS and uses between 2.2 and 4.2 times less space.

These experiments were run on a Decstation 5000/200 with 64 meg, running Mach 2.6. Both compilers were run at the highest level of optimization and the intermediate C files generated as well as the run-time libraries were compiled with all optimizations turned on. CParaOPSS is comparable in speed with ParaOPSS [60] which, in turn, has been shown to be comparable in

Programs	CIIOPS5 time	PPL time	speedup	PPL space	CIIOPS5 space	space ratio
rubik	14.15s	5.9s	2.38	2 meg	4.5 meg	2.25
tourney	6.53s	3.9s	1.67	2 meg	7 meg	3.5
gen.tsp	12.50s	3.1s	4.03	2 meg	8 meg	4.0

In the table, CIIOPS5 stands for CParaOPSS

Table 4.1: Comparison of uniprocessor PPL and CParaOPSS for fixed data set benchmarks

Programs	CIIOPS5 time	PPL time	speedup	PPL space	CIIOPS5 space	space ratio
make-teams	5536s	2714s	2.04	3 meg	6.6 meg	2.2
clusters	1286s	14.3s	89.93	3 meg	12.5 meg	4.2
airline-route	7678s	401s	19.15	14 meg	30.5 meg	2.18

In the table, CIIOPS5 stands for CParaOPSS

Table 4.2: Comparison of uniprocessor PPL and CParaOPSS for variable data set benchmarks

speed with ops5c [82]. Several unsuccessful attempts were made to obtain ops5c for a direct comparison with PPL. These experiments are the first to use uniprocessor implementations of optimized sequential programs as the baseline for measuring parallelism in production system programs. Previous research efforts have either used multiprocessor implementations running on a single processor as the baseline or have not used optimized sequential programs or both. We have already seen that optimization of sequential programs can yield up to 18 fold speedup. Results presented in the next chapter indicate that for tasks as fine-grained as those occurring in production system programs, the parallelization overhead can be as large as a factor of 2.5.

Another common limitation of previous research efforts that studied parallelism in production system programs is that they limited themselves to particular sections or kernels of the programs and not the full implementation. In particular, most research efforts have focussed their research on parallelizing the match phase [2, 14, 38, 42, 51, 61, 84, 90, 105] and have ignored the costs of the select and act phases. To achieve scalable parallelism, *all* phases of the implementation must be parallelized. An inefficiency in any one of the phases will place an Amdahl's law limitation on the overall speedup.

All the benchmarks used in these experiments run for long periods of time - the largest run corresponds to over 50 billion instructions. This limits the variance due to operating-system-specific initialization costs, e.g. creating processes.

The two PPL implementations, uniprocessor and multiprocessor, share most of the code, both compiled code and run-time library code, and differ only in their support for parallelization. This eliminates potential distortion due to differences in compilation strategies and/or run-time

support.

These experiments had two orthogonal parameters – the size of the data set and the number of processors being used. Even though the PPL implementation allows the number of task stacks to be varied between one and the number of processors, for these experiments, the number of task-stacks was fixed at the number of processors. Multiple task-stacks have been shown to significantly outperform a single task-stack for parallel execution of production system programs [42].

Each experiment consists of running one of the versions of the program, and feeding the trace generated into the simulator. Close to 3000 such experiments were run. Both PPL implementations were modified to generate a trace of their execution which was used to drive the simulation. Initially, the trace generated was stored on disk and reused for several simulations. However, as the size of the data sets, and hence the length of the program runs, grew it was not possible to store the traces on disk (the largest trace generated was about 650 Megabytes). Instead, the simulator was run concurrently with the benchmark program and the trace was fed to it over a Unix socket. The simulations were run on a large number of workstations including Decstation 5000/200s and the Alpha AXP based Decstation 3000/400s. The next section describes the simulator.

4.3 Simulator

Previous studies exploring parallelism in production systems have been based on simulators that were limited in various ways. Several of the simulators have been based on very simple cost models [5, 50, 56, 65, 81, 90, 92, 101, 104]. Others have been based on average case data [51]. These simulators did not take many overheads into account and often ignored variations in the cost of *msa* cycles. Gupta [38] reported results from a simulator based on an accurate and detailed cost model. However, his simulator simulates only the match phase, uses an average case assumption to determine the cost of processing a token and does not include the costs of memory management. Furthermore, the trace used to drive the simulator does not contain enough information to create a complete dependency graph; for tasks with two parents, for example tokens in the Rete network, it contains dependency information only for the parent that occurred later. The simulator used in these experiments simulates the complete execution of a production system program based on a fine-grain and accurate cost model and uses a detailed trace that contains all the dependency information. Section 4.3.1 describes the structure and the operation of the simulator. Section 4.3.2 discusses its limitations and Section 4.3.3 argues about the validity of the results.

4.3.1 Structure and operation of the simulator

The simulator assumes a shared memory model with uniform access time. It measures cost in number of instructions executed. It is event-driven. Like Proteus [10], each processor is simulated for a complete task before yielding the simulator to another processor. This allows fast simulation of long-running programs. It simulates all phases of production system execution. The structure of the simulator mirrors that of the implementation and reuses most of the code. The simulator is based on a detailed basic-block level cost model. It is driven by a compact trace which contains information about the entire execution of the program. Simulation of the program is done in cycles. Each cycle simulates the operations between successive barrier synchronizations. The input to the simulator consists of: (1) a trace of the operations performed during the execution of the program (2) a description of the program in terms of the productions, production sets and Rete network nodes (3) a detailed basic-block level cost model (4) a description of the machine configuration to be simulated. The output of the simulator consists of various program-level and processor-level statistics as well as some information about individual cycles. Section 4.3.1.1 describes the trace and the mechanism used to generate it. Section 4.3.1.2 describes the cost model and how it was generated. Section 4.3.1.3 describes the operation of the simulator.

4.3.1.1 Trace

The trace used to drive the simulator is complete in that it contains information about all operations that take place during the execution of the program. However, the trace format is compact and contains information only about selected events from which information about the rest of the events can be generated. In particular, it traces only the information that is needed to decide branch directions and iteration counts in the simulator. This is similar to the *Abstract Execution* technique [68] proposed by Jim Larus to reduce the length of traces. Since the simulator is closely allied to the implementation, it can regenerate the information itself, there is no need for auxiliary programs like those generated by AE. Figure 4.6 contains the C struct declaration for a trace record.

Since the trace cannot contain pointers, links between tasks are indicated using *activation_ids*. Every task that can generate successors has a unique *activation_id*. Trace records for successor tasks refer to their parents using these *activation_ids*.

Foreign function calls present special problems for tracing. Since the functions called can perform arbitrary computation, it is not possible to trace their execution within this framework. To determine the cost of a foreign function call, the tracing library makes use of a Decstation-specific timing board that contains a 32-bit bus cycle counter. It uses external memory support in Mach 3.0 to map this counter into memory. The cost of timing a function call is equal to two trips to the memory, which is about 20 processor cycles. Since most foreign function calls are

```

typedef struct PPL_TRACE_RECORD_REC {
    int proc_num_and_record_type, /* low 16 bits for rec_type */
    union {
        struct {
            int tuple_time_tag, size, activation_id, num_of_tests,
            int num_of_successors_and_add_or_delete;
            /* bit_0 = add/delete, bit_1-31 = #successors */
        } alpha;
        struct {
            int left_parent_id, right_parent_id, activation_id, node_id;
            int attributes;
            /* bit_0=add/delete, bit_1=left/right, bit2-3 for scheduler,
             * bit_4-31 num_of_successors */
            int test_values;
        } beta;
        struct {
            int left_parent_id, right_parent_id, prod_id;
            int pset_id_and_add_or_delete;
            /* bit_0 = add/delete, bit_1-31 = pset_id */
            int *timetags;
        } pnode,
        struct /* header record for a select-match-act cycle */
            int cycle_num, record_id;
    } select;
    struct {
        int activation_id, pset_id, prod_id;
    } fire;
    struct {
        int parent_id; /* activation id of fire record */
        int type;
        union {
            struct { int timetag; } make;
            struct { int timetag_new, timetag_old; } modify;
            struct { int timetag_new, timetag_old; } copy;
            struct { int timetag; } remove;
            struct { long cost; } call;
        } info;
    } action;
    struct {
        int type;
        union {
            struct { int num_of_values; } substr_call;
            struct { long cost; } function_call;
        } info;
    } value_item;
    struct { int parent_id, timetag, } external_make;
    } info;
} ppl_trace_record_rec, *ppl_trace_record_ptr;

```

Figure 4.6: C struct declaration for a trace record

significantly longer (of the order of tens of thousands of instructions), this cost is negligible. Since this is a free-running counter, there is no way to associate costs with processes. Therefore, to eliminate distortion due to extraneous processes being scheduled, the environment on the workstation on which the traces were generated was severely curtailed. Running the simulator concurrently, as done in other experiments, is not suitable for experiments require tracing foreign function calls. A one gigabyte disk was attached to the workstation to store the traces. Unix buffering was disabled. Trace records were logged in memory and were written out only when the counter was not in use.

The trace is supplemented by a static trace data file generated by the compiler which contains detailed information about individual productions, production sets, and Rete network nodes. For details about this data file as well as further information about the trace format, see Appendix A.

Even though the trace format and the tracing library support traces from multiprocessor execution, the traces used in the experiments were taken from uniprocessor runs. For uniprocessor simulations, traces were generated using the baseline versions of the programs. For simulation of automatically parallelized sequential programs on multiprocessors, traces were generated using the parallel-match versions of the programs running on a single processor. For simulation of parallel programs on multiprocessors, traces were generated using the parallel versions of the programs running on a single processor.

Using a uniprocessor trace for multiprocessor simulations introduces a distortion. Consider the Rete network and the pair of tuple-space modifications in Figure 4.7. If the deletion happens before the addition, then T7 has to be compared only with T1 and T3. Otherwise, it has to be compared with T1, T2 and T3. Furthermore, different orders of processing the tuple-space modifications can lead to different number of tokens being generated in the Rete network. For example, in Figure 4.7, if T7 is added before T2 is deleted and T7 matches T2, a successor token, (T2,T7), is generated. However, if deletion of T2 happens before T7 is added, no successor task is generated when T7 is subsequently added. Therefore, the number of tasks and the exact length of each task depends on the order in which tuple-space modifications and the token they generate are processed. In a uniprocessor environment, all tasks are processed in a depth-first manner but in a multiprocessor environment, the order depends on the number of processors and their relative speeds. Even with this disadvantage, uniprocessor traces are still the best possible option for the following reasons:

- This distortion will arise every time a trace from one configuration is used to simulate a multiprocessor of a different configuration. Therefore, unless we have parallel machines for all the configurations we would like to simulate, this distortion is inevitable. In which case, using the simplest possible configuration is the best option.
- Even if we had parallel machines for all the configurations we would like simulate, the tracing process itself introduce distortion in the execution of the program and, thus,

would change the order in which the tokens are processed. A uniprocessor execution is free of this distortion.

Previous research efforts that have experimented with changes in order of token processing report that the difference in execution time has been under 10% [111]. The simulator supports various token ordering policies. Experiments conducted as a part of this investigation have indicated that the difference in execution time between a depth-first order and a breadth-first order is under 5%.

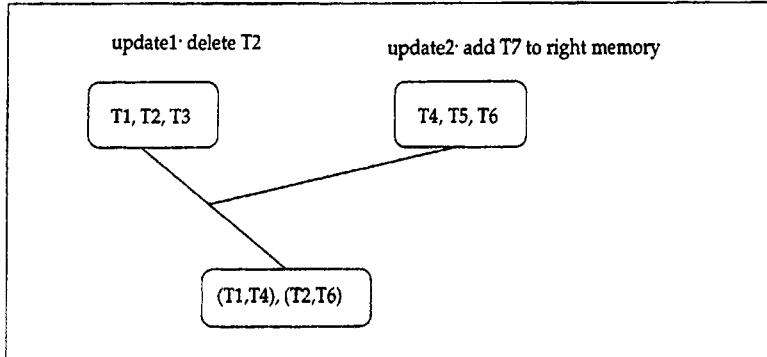


Figure 4.7: Effect of processing order on token cost

4.3.1.2 Cost model

The cost model consists of costs, in terms of the number of instructions needed, for all the primitive operations in the execution of a PPL program. Examples of primitive operations include individual α tests, hashing an integer value (for looking up a token in the hashtables), caching the values being matched (before traversing a hash bucket), traversing each link in the list of tokens in a hash bucket, extracting value from a token, performing a single β test between two tokens, adding an instantiation to a list, extracting the timetag from a tuple, initializing a field, creating a task structure, freeing a task structure and so on. Each primitive operation consists of straightline code - either a single basic block or a small number of basic blocks linked by high probability branches. There are 287 such operations. A complete list can be found in Appendix B.

The cost model was not hardwired into the simulator. Instead, costs for all the primitive operations were loaded along with the static trace data file and a description of the machine

configuration All simulations for the experiments reported in this dissertation used a cost model based on the MIPS R3000 instruction set. The cost model was generated in the following way:

1. The C code for the run-time library and the productions was compiled to optimized object code using `gcc -O`. It was possible to do this for the productions because of the highly stylized nature of the C code generated for them. Assertions and other debugging code were conditionally compiled out.
2. The object code was disassembled and the code for individual routines was extracted using the Ultrix disassembler.
3. The assembly code for individual routines was passed through a program that discovered and marked the basic blocks in the code. Figure 4.8 shows the result for the routine `ppl.add_to_tuple_space()` from the multiprocessor implementation.
4. The basic blocks were manually mapped back to the original C code and costs were associated with individual primitive operations.

Since individual costs are extracted from optimized object code that would actually run, they are accurate. There are separate cost models for uniprocessor and multiprocessor implementations. Figure 4.9 shows the code for `ppl.add_to_tuple_space()` taken from the uniprocessor implementation.

To see how the cost of a primitive operation is calculated, consider the disassembled code in Figures 4.8 and 4.9 and the corresponding C code in Figure 4.10. Both cases contain two conditional branches, one at the end of the first basic block and the second at the end of the third basic block. Both these branches test if the tuple-space is empty. Since the tuple-space is almost never empty,¹ the loop branches can be assumed to be always taken. With this assumption, the cost of adding a tuple to the tuple-space is 35 instructions for the multiprocessor implementation and 20 instructions for the uniprocessor implementation.

4.3.1.3 Operation

As mentioned above, the input to the simulator consists of the trace, the static trace data file, the cost model and a configuration file. The configuration file specifies the machine configuration to be simulated (number of processors, number of task-stacks, the kind of hashtables used, the output statistics to be computed etc). Appendix C describes the configuration file structure.

¹This is not surprising since nothing happens in a production system program with an empty tuple-space!

```

MULTIPROCESSOR:
ppl_add_to_tuple_space:
0x0: 27bdffe8 addiu sp,sp,-24           | begin 1 (19 cycles)
0x4: afb00010 sw s0,16(sp)
0x8: 008c8c21 move s0,a0
0xc: 8f848020 lw a0,-32736(gp)
0x10: afbff0014 sw ra,20(sp)
0x14: 0c000000 jal ppl_acquire_lock_private
0x18: 00000000 nop
0x1c: 8f828010 lw v0,-32752(gp)
0x20: 00000000 nop
0x24: 8c430000 lw v1,0(v0)
0x28: 00000000 nop
0x2c: ae030010 sw v1,16(s0)
0x30: ac500000 sw s0,0(v0)
0x34: 8f838018 lw v1,-32744(gp)
0x38: 00000000 nop
0x3c: 8c620000 lw v0,0(v1)
0x40: 00000000 nop
0x44: 14400002 bne v0,zero,0x50
0x48: 00000000 nop
0x4c: ac700000 sw s0,0(v1)           | end 1
0x50: 8e020010 lw v0,16(s0)           | begin 2 (1 cycles)  end 2
0x54: 00000000 nop
0x58: 10400002 beg v0,zero,0x64
0x5c: 00000000 nop
0x60: ac500014 sw s0,20(v0)           | begin 3
0x64: 8f848020 lw a0,-32736(gp)       | begin 4 (1 cycles)  end 4
0x68: 00000000 nop
0x6c: 0c000000 jal ppl_release_lock_private
0x70: ae000014 sw zero,20(s0)           | begin 5
0x74: 02002021 move a0,s0
0x78: 0c000000 jal ppl_match_tuple
0x7c: 00002821 move a1,zero
0x80: 8fbff0014 lw ra,20(sp)
0x84: 8fb00010 lw s0,16(sp)
0x88: 03e00008 jr ra
0x8c: 27bd0018 addiu sp,sp,24           | end 5

```

Figure 4.8: Disassembled multiprocessor code for ppl.add_to_tuple_space()

```

UNIPROCESSOR:
ppl_add_to_tuple_space:
0x0: 8f828010 lw v0,-32752(gp)      | begin 1 (8 cycles)
0x4: 27bdffe8 addiu sp,sp,-24
0x8: afbf0010 sw ra,16(sp)
0xc: ac82000c sw v0,12(a0)
0x10: 8f828018 lw v0,-32744(gp)
0x14: af848010 sw a0,-32752(gp)
0x18: 14400002 bne v0,zero,0x24
0x1c: 00000000 nop
0x20: af848018 sw a0,-32744(gp)
0x24: 8c82000c lw v0,12(a0)
0x28: 00000000 nop
0x2c: 10400002 beq v0,zero,0x38
0x30: 00000000 nop
0x34: ac440010 sw a0,16(v0)
0x38: ac800010 sw zero,16(a0)
0x3c: 0c000000 jal ppl_match_tuple |
0x40: 00002821 move a1,zero
0x44: 8fbff010 lw ra,16(sp)
0x48: 27bd0018 addiu sp,sp,24
0x4c: 03e00008 jr ra
0x50: 00000000 nop
                                         | end 5

```

Figure 4.9: Disassembled uniprocessor code for `ppl_add_to_tuple_space()`

The simulator simulates each select-act-match cycle, that is the operations between successive barrier synchronizations, separately. The trace contains complete information about the dependencies between tasks. In particular, it records information about both parents of tokens. Traces used in previous studies recorded information about only the parent that occurs later. As mentioned in Section 4.3.1.1, dependency information is encoded using the *activation.ids* of tasks. However dependencies that span barrier synchronizations are clipped since the tasks involved are simulated in different cycles.

The simulator collects a wide variety of statistics. Figure 4.11 shows an abbreviated version of the output file for one of the experiments.

4.3.2 Limitations of the simulator

The sources of inaccuracies in the simulator are:

- The simulator assumes a uniform access memory model - all instructions are assumed to cost the same. Modern architectures do not support such a model and good memory

```

void ppl_add_to_tuple_space(tuple)
ppl_tuple_ptr tuple;
{
    /* macro -- expands to whitespace for uniproc impl */
    ppl_acquire_lock(tuple_space_lock);

    /* cons tuple onto tuple-space list. ppl_ref_value() and ppl_update_ref()
     * are macros which expand to indirect operations for multproc impl and
     * direct operations for uniproc impl */
    tuple->next = ppl_ref_value(tuple_space_head);
    ppl_update_ref(tuple_space_head,tuple);

    /* checks if the tail is NULL, happens iff tuple-space is empty */
    if (ppl_ref_value(tuple_space_tail) == NULL)
        ppl_update_ref(tuple_space_tail,tuple);

    /* tuple-space is a doubly linked list. link back from next tuple,
     * if there is a next tuple. */
    if (tuple->next != NULL)
        tuple->next->prev = tuple;

    /* this is the first tuple */
    tuple->prev = NULL;

    /* macro -- expands to whitespace for the uniproc impl */
    ppl_release_lock(tuple_space_lock);

    /* invoke rete match code */
    ppl_match_tuple(tuple,PPL_DIRIN); /* PPL_DIRIN == 0 */
}

```

Figure 4.10: C code for `ppl.add_to_tuple_space()`

subsystem performance is important for achieving good speedups. However, as far as scalable parallelism is concerned, memory subsystem performance is a second order effect (an important second order effect but a second order effect nevertheless). Overcoming the program-independent bound on available parallelism is the primary issue and is the focus of this dissertation. Once this issue has been addressed, the memory subsystem performance will become the most important issue. As shown in Table 4.1, memory usage of programs compiled with PPL is between 2 and 4 times less than those compiled with previous compilers. This should improve their memory locality and reduce the effect of the memory subsystem.

```
Length of trace = 1.44017e+08 bytes
Total time = 4.92692e+08 processor cycles
Processor id 0: utilization = 99.83
Processor id 1: utilization = 98.68
Processor id 2: utilization = 98.90
Average utilization = 99.14
Number of instantiations fired = 19558, Number of cycles = 1560
Number of firings per firing cycle = 12.5533
Number of wm deletions = 19657
Number of wm additions = 21013
Number of wm changes = 40670
Number of wm changes per active cycle = 26.0872
Time breakdown:
Time in multiprocessor code:
Time in alpha activations: 6.72317e+06
Time in beta activations: 8.39499e+08
Time in pnode activations: 4.96327e+07
Time in rhs : 1.12228e+07
Time in function calls : 0
Time in scheduling: 5.22371e+08
Time in memory reclaiming: 3.3776e+07
Time in fire: 473900
Time in uniprocessor code
Time in select: 1.84899e+06
Time in overheads. 208795
Task information: total tasks = 4442754
Alpha tasks = 40670 (0.92 %)
Add beta tasks = 2074873 (46.70 %), delete beta tasks = 2036588 (45.84 %)
Conj add beta tasks = 97775 (2.20 %), conj delete beta tasks = 97775 (2.20 %)
Add pnode tasks = 32750 (0.74 %), delete pnode tasks = 13192 (0.30 %)
Refracted instantiations deleted = 19557 (0.44 %)
Efficiency of instantiation generation = 51.80 %
Conj add pnode tasks = 5008 (0.11 %), conj delete pnode tasks = 5008 (0.11 %)
rhs tasks = 19558 (0.44 %)
```

Figure 4.11: Abbreviated simulator output for one of the experiments

- Traces taken from uniprocessor runs are used for all simulations. As discussed in Section 4.3.1.1, this leads to inaccuracies in the number of tasks and the costs of individual tasks. There is no reasonable way of avoiding this inaccuracy. Previous research efforts that have experimented with changes in order of token processing report that the differences have been under 10% [111]. The simulator supports various token ordering policies. Experiments conducted as a part of this investigation have indicated that the difference in execution time between a depth-first order and a breadth-first order is under 5%.
- Contention for shared resources is not taken into account. As described in Section 3.4.5, considerable effort has been devoted to minimizing contention for shared resources. The only resources for which there might be significant contention are the hashtables that store the contents of the Rete memory nodes. If this becomes a problem, the hashtable size can be easily increased.

As mentioned above, several approximations have been made in the construction of the simulator. However, as discussed in an earlier part of this section, this is the most accurate simulator that has been used for studying production system programs and is the only simulator that simulates the *all* operations in a production system program. I believe that it accounts for most of the important costs and variations.

4.3.3 Validity of the simulator

In any simulation based study, it is necessary to establish the validity of the simulator in some way. The best way would be to run the programs on a parallel machine of suitable configuration and compare the results with the results from the simulator. But it is the lack of suitable machines that lead to the construction of the simulator in the first place. In absence of direct confirmation, the belief in the validity of the results is based on the following facts:

- The simulator can be used to predict the running time on a uniprocessor with reasonable accuracy. Since the simulator does not model the memory subsystem, a constant (or nearly constant) ratio between the wall clock time on an actual uniprocessor and the instruction count generated by the simulator is unlikely. Especially for data sets of widely varying sizes. However, an analysis of the results shows that for each benchmark, this ratio decreases linearly with the increase in the data set size. The rate of decrease is steady enough to be used to predict the running time for larger data sets. Table 4.3 shows the predicted time and the actual wall clock time for several benchmarks and data sets. The wall clock time is from executions on a Decstation 5000/200 running Mach 2.6, with 64 megabytes of memory.

Program	data set	predicted time	actual time
circuit	250 bits	46.8s	47.6s
circuit	275 bits	51.7s	53.4s
life	60 reps	105.3	106.5s
life	70 reps	132.1s	141.8s
hotel	6 floors	1519.4s	1536.6s
hotel	7 floors	2791.8s	2853.5s
spam	sf4917	1509.5s	1721.1s

Table 4.3: Comparison of predicted and actual running times

- The simulator detected inefficiencies in the implementation and accurately predicted the magnitude of the improvements for alternative implementations. Based on results from previous research, the occurrence of conjugate instantiations was assumed to be low and accordingly the implementation used a single linked list per production to hold them. Simulations indicated that the number of conjugate instantiations grew roughly with the growth in the number of tuple-space changes per *msa* cycle and the number of processors. In simulations of *waltz*, the cost of handling conjugate instantiations grew to dominate the execution. Uniprocessor execution of the *parallel* version of *waltz* backed up this discovery. The simulator also accurately predicted the speedup achieved (for uniprocessor execution) by replacing the single per-production linked list by a per-production hashtable. A similar inefficiency was found in the handling of refracted instantiations (instantiations that have already been fired)
- The simulator is quite close to the implementation and shares most of the code. Along with the detailed and accurate cost model, this indicates that the simulator should be closely modeling the implementation
- Speedups for the *parallel-match* versions of most programs are comparable with those reported by efforts to automatically parallelize production system programs. One of the programs, *hotel*, indicates large amounts for parallelism for the *parallel-match* version. The reason for this is discussed as a part of the analysis presented in the next chapter.

Chapter 5

Parallelism Experiments: Results, Analysis and Observations

The goal of these experiments was to test three hypotheses. First, that, in general, there is no program-independent bound on the speedup in parallel production system programs. Second, that speedups in parallel production system programs can scale with data. Third, that production sets and parallel productions are effective for the expression of parallelism in production system programs. This chapter presents and analyzes the results of the experiments with the aim of validating these hypotheses. To show that there is no program-independent bound on the speedup in parallel production system programs, it presents speedups for the full benchmark suite. To show that the speedup can scale with data set size, it shows how the speedup varies with data set size. To show that production sets and parallel productions are effective for the expression of parallelism, it compares the speedups achieved by two versions of the benchmark programs, one that uses these constructs and the other that does not. It analyzes the results to identify the factors that limit speedups in parallel production system programs. The chapter concludes with some observations from the experiments, including programming idioms for parallel production system languages and practical advice for parallelizing sequential production system programs.

5.1 Speedups

This section presents speedups for the full benchmark suite, the goal being to validate the hypothesis that there is no program-independent bound on speedups in parallel production system programs. The parameter space for the experiments is two-dimensional, data set size being one dimension and number of processors the other. Results presented in this section assume a configuration with 100 processors. This configuration is large enough to demonstrate

the variation between the speedups achieved by different benchmarks. Selection of coordinates along data-set-size dimension is more difficult since the size of the tuple-space can vary greatly depending on the programming style. Instead, a reasonably large data set was independently selected for each benchmark. Table 5.1 contains the selected values of the data set parameters for all the benchmarks. For reference, it also contains the number of tuples in each data set. Results presented in Section 5.2 show that for three of the benchmarks, *circuit*, *life* and *waltz*, the growth in the speedups with data set size is low beyond these parameter values indicating that these values are suitable for the comparison. These instances of the benchmarks will be referred to as the *comparative instances* and the whole set will be referred to as the *comparative suite*.

Program	<i>circuit</i>	<i>life</i>	<i>waltz</i>	<i>hotel</i>	<i>spam</i>
Data set	200 bits	70 repetitions	120 repetitions	7 floors	dc36809
Number of tuples	400	1051	6996	10873	367
Uniproc time (10^6 instrs)	262.1	589.1	426.7	12477.6	23862.5

Table 5.1: Data sets for the comparative suite

Figure 5.1 shows speedups achieved by the comparative suite. These speedups are computed using the sequential versions of the benchmarks as baselines. As described in Section 4.2, the sequential version of a benchmark program is an efficient uniprocessor implementation of the program and is built by compiling the sequential version of the program with the PPL compiler targeted towards uniprocessors.

To explore the growth of speedup beyond these data set sizes, additional experiments, with larger data sets and machine configurations were selectively conducted (as justified by the results). Table 5.2 shows the highest speedup achieved for each benchmark along with the corresponding data set size and the machine configuration.

Program	<i>circuit</i>	<i>life</i>	<i>waltz</i>	<i>hotel</i>	<i>spam</i>
Highest speedup	29.6	23.6	17.9	115.3	52.3
Number of processors	100	100	100	200	100
Data set size	275 bits	80 reps	120 reps	10 floors	dc36809

Table 5.2: Highest speedups achieved.

These results indicate that there is no discernible limit on speedups in parallel production system programs. Furthermore, analysis presented in Section 5.2 concludes that for two of the benchmarks, *hotel* and *spam*, increasing the data set size further can lead to even larger speedups.

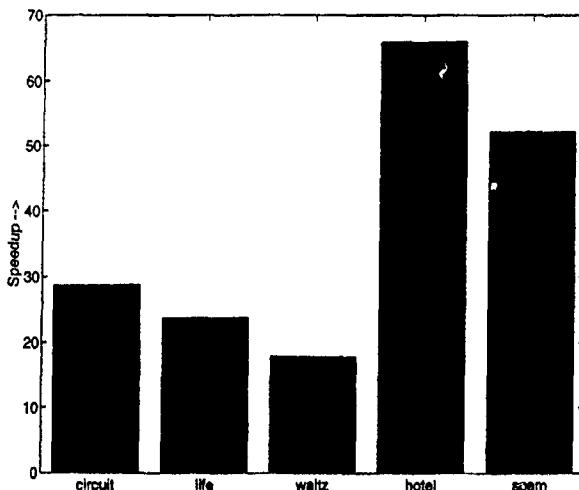


Figure 5.1: Speedups for the comparative suite.

There is significant variance between the speedups achieved by the benchmarks. Of the five benchmarks, *hotel* and *spam* achieve large speedups (> 50 fold for 100 processors) whereas the other three achieve much smaller speedups. The following subsection analyzes the benchmarks and identifies the factors that limit speedups in parallel production system programs.

5.1.1 Analysis

There are three major limitations on speedups in parallel production system programs: parallelization overheads, non-parallelizable loops and dependencies between tasks.

5.1.1.1 Parallelization overheads

Effective load-balancing for parallel production system programs requires fine-grain decomposition. Individual tasks can often be as small as a few hundred instructions. Table 5.3 shows the average task size and the parallelization overhead for the comparative suite. The parallelization overhead is computed by taking the ratio of the *nominal* speedup and the *real*

speedup. Nominal speedup is speedup with respect to the parallel version of the program running on one processor which includes the cost of running the program in parallel. The real speedups, on the other hand, are computed with respect to the sequential version which includes no parallelization costs. All the speedups shown in this chapter are real speedups. The major component of the parallelization overheads shown in Table 5.3 is the cost of scheduling tasks which includes creation/deletion of task records, addition/removal of task records from task stacks. Other components include costs of locking and barrier synchronization. The data in Table 5.3 indicates that the average task size is small for all the three low-speedup benchmarks. Accordingly, they incur large parallelization overheads. On the other hand, the average task size is large for both high-speedup benchmarks resulting in very low parallelization overheads. Figure 5.2 compares the nominal speedups for the comparative suite. It shows that after taking the parallelization overhead into account, the speedups for *circuit* and *life* are comparable to the speedups for *hotel* and *spam*. In fact, the speedup for *circuit* is higher than both of them. However, the speedup for *waltz* remains relatively low.

Program	<i>circuit</i>	<i>life</i>	<i>waltz</i>	<i>hotel</i>	<i>spam</i>
Task size (instrs)	92.2	162.0	440.5	23858.3	47820.1
Parallelization overhead	2.6	1.9	1.5	1.0	1.0

Table 5.3: Parallelization overhead

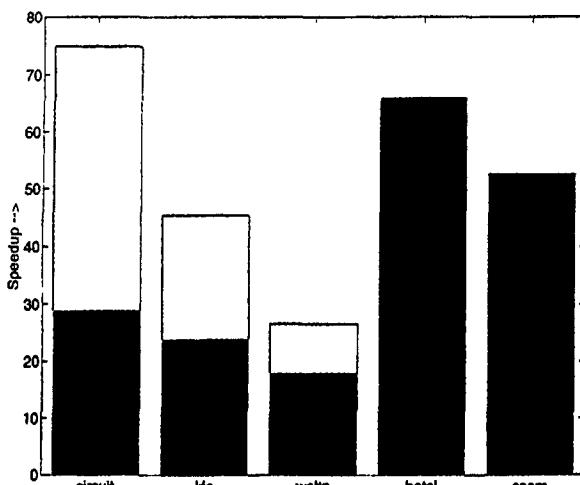
Table 5.4 shows the number and average size of the various kinds of tasks in the benchmark programs. It shows that in benchmarks with high parallelization overheads, tasks from the match phase dominate the execution and that they are usually significantly smaller than the tasks from select or act phases. Therefore, match tasks are the major cause of high parallelization overheads in these benchmarks.

Program	<i>circuit</i>	<i>life</i>	<i>waltz</i>	<i>hotel</i>	<i>spam</i>
Match tasks (millions)	2.6028	3.3818	0.7697	0.3805	0.1935
Select tasks	62293	74172	156968	142193	304,174
Action tasks	20275	15258	59524	7991	98,116
Average match task size	83.2	150.1	183.6	32221.2	221.9
Average select task size	340.2	618.3	1539.6	1414.3	365.9
Average action task size	387.2	480.8	540.1	1203.1	241282

Task sizes are in number of instructions.

Table 5.4: Distribution and average size of tasks in the comparative suite

The problem of high parallelization overheads can be alleviated by grouping match tasks. An interesting way of grouping match tasks is described in Chapter 6 which introduces collection-



Shaded bars correspond to *real* speedups, unshaded bars correspond to *nominal* speedups.

Figure 5.2: Real and nominal speedups for the comparative suite

oriented match algorithms. These algorithms group the tokens in each memory node into equivalence classes based on the values that are going to be tested subsequently.

5.1.1.2 Non-parallelizable loops

As mentioned in Section 4.1, it is not possible to parallelize some loops due to the inherently sequential nature of the computation. For example, the loop that prints the matrix in *life* or the loop that finds and merges overlapping functional areas in *spam*. In such loops, only one instantiation is fired per *msa* cycle. This limits the speedup in two ways – it limits the speedups in the match phase and it forces sequential execution in the action phase.

Low match speedups

Figure 5.3 shows a non-parallelizable doubly nested loop from *life*. This loop prints the entire matrix of cells. Since the cells have to be printed in a fixed order, this loop cannot be parallelized. Each firing modifies only one tuple, the *print-coordinates* tuple which

holds the coordinates of the next cell to be printed. The *msa* cycles that are a part of this loop have few match tasks and hence low concurrency in the match phase. Since these cycles can be sped up only by small factors, the time spent in such cycles limits the overall speedup as an Amdahl's law effect.

```

(p print-next-cell-in-row ; inner loop
  (print-coordinates ^x <x> ^y <y>)
  (cell ^x <x> ^y <y> ^status <status>)
  -->
  (write <status>)
  (modify 1 ^x (<x> + 1))) ; go to next cell in this row

(p switch-rows ; outer loop
  (print-coordinates ^x <x> ^y <y>)
  -(cell ^x <x>) ; no more cells in current row
  -->
  (write "\n")
  (modify 1 ^x 0 ^y (<y> + 1))) ; go to first cell in next row

(p finalize-print
  (print-coordinates ^y <y>)
  -(cell ^y <y>) ; no more rows
  -->
  (remove 1)) ; terminate outer loop

```

Figure 5.3: Non-parallelizable doubly nested print loop from *life*

Table 5.5 contains information about the average *msa* cycle size for the comparative suite. Of these, *life* has the smallest average cycle size. The primary reason for this is the non-parallelizable loop shown in Figure 5.3. In the basic pattern used as building block for *life* data sets, only two out of the thirty cells change state. The status of the other twenty-eight cells remains fixed throughout the run. However, all cells are printed. As a result, the number of iterations of the non-parallelizable print loop grows much faster than the number of iterations of the parallelizable simulation loop. For the comparative instance of *life* (70 repetitions of the basic pattern), the simulation loop takes 200 *msa* cycles and the print loop takes 1058 cycles.

Program	circuit	life	waltz	hotel	spam
Average firings/cycle	99.8	12.1	1920.1	24.7	28.7
Average tasks/cycle	13064.3	2743.6	29454.5	16083	145.8
Average cycle size (million instrs)	1.2846	0.4675	13.335	38.3926	6.9814

Table 5.5: Average size of *msa* cycles

Sequential actions

While firing only one instantiation per *msa* cycle limits the speedup in the match phase, it totally eliminates parallelism in the action phase. Actions corresponding to each instantiation are executed sequentially. This increases the fraction of time spent in sequential code which limits the overall speedups as an Amdahl's law effect. This factor can be particularly important if the actions being executed include calls to expensive foreign functions. Such a situation occurs in the third phase of *spam*. As mentioned in Section 4.1.5, this phase contains two non-parallelizable loops that merge links between image regions and coalesce overlapping functional areas (e.g. runways). Figure 5.4 shows stripped down versions of the productions that implement the second loop. This loop cannot be parallelized since a functional area may overlap with several other functional areas and the desired result depends on a particular order being followed. Note that the actions include a call to *OPS.overlap* which attempts to determine whether two functional areas overlap and if so to what degree. The average cost of calls to this routine is about 61000 instructions and there are 2780 such calls, one per iteration, for the dc36809 data set. This limits the speedup achieved by *spam* to little over 50 fold even though the average *msa* cycle is large and there is virtually no parallelization overhead. However, *spam* shows larger speedups than *life* because its parallelizable loops are much larger than its non-parallelizable loops. For dc36809, the fully parallelizable loops of first and second phase have 17,100 and 35,611 iterations respectively whereas the non-parallelizable loops have only 621 and 2780 iterations respectively.

Since problem requirements force certain loops to be sequential, it is not possible to eliminate non-parallelizable loops. However, sequential loops in production system languages pay an additional price for using matching even though the control-flow is fixed. In a tight loop, like the loop in *life*, this cost could dominate the cost of the real computation in the loop. This cost could be eliminated by collecting the items to be operated on in the loop and passing the collection to a routine written in a procedural language. This would reduce the fraction of time spent in sequential code. Chapter 7 discusses a production system language which supports automatic generation of collections of tuples as well as aggregate operations on these collections.

```

(p FA**attempt-generalization ; select two areas of the same type
  (functional-area ^convex-hull {<h1> <> nil} ^type <>)
  (functional-area ^convex-hull {<h2> <> <h1> <> nil} ^type <>)
  -->
  (make fa-score ^fa-1 <h1> ^fa-2 <h2> ^overlap OPS_overlap(<id1> <id2>)))

(p FA**generalization-successful , there is over 90% overlap
  (fa-score ^overlap >= 90 ^fa-2 <hull2>)
  (functional-area ^convex-hull <hull2>)
  -->
  (remove 1)
  (modify 2 ^flag inactive)) ; merge the second area into the first

(p FA**generalize-failed , less than 90% overlap
  (fa-score ^overlap < 90)
  -->
  (remove 1)) ; do not merge the areas

```

Figure 5.4: Non-parallelizable loop from third phase of *spam*

5.1.1.3 Inter-task dependencies

Inter-task dependencies in parallel production system programs arise from the structure of the Rete network. The Rete network attempts to restrict the number of potential matches by specifying a fixed order for the tests in a production. Therefore, all the tokens (match tasks) that are involved in the generation of a single instantiation must be executed in a fixed sequence. Even though a large number of tasks might be available in a *msa* cycle, it can happen that only a small number can be executed at any given time. This limits the speedup in such cycles. This is referred to as the *chaining* effect. This effect becomes a serious limitation in cycles with a small number of tasks, for example, in cycles in a non-parallelizable loop. It can also become important if some of the productions are substantially longer than others. In such cases, chaining in these productions can lead to a long period of low parallelism at the end of the cycle. The chaining effect was identified in Anoop Gupta's thesis [38] and has been shown to be one of the major limitations on speedups for automatic parallelization of OPS5[42] as well as for parallel implementation of the match phase in Soar[113]. Chaining effect is likely to

occur if the variation in the lengths of productions is high, since some of the instantiations being generated are likely to be much longer than others. However, chaining effect sequentializes only the tokens involved in the generation of the same instantiation. If a large number of instantiations are generated per cycle, presence of chaining is not a serious limitation. Table 5.6 shows the mean and standard deviation for the number of conditions per production for all the benchmarks. Given the large standard deviations for *life* and *hotel*, chaining effect can be expected to occur in these benchmarks. Table 5.7 shows the number of instantiations generated per cycle for comparative suite. It shows that relatively few instantiations are generated per cycle in *life*. Therefore, chaining is likely to be a major limitation on speedups in *life*. Chaining is less of a limitation in *hotel* since a relatively large number of instantiations are generated per cycle.

Program	circuit	life	waltz	hotel	spam
Average conditions/production	5.49	5.3	3.08	3.32	2.56
Standard deviation	1.98	2.76	0.86	2.7	0.89
Ratio	0.36	0.52	0.28	0.88	0.35

Table 5.6: Mean and standard deviation for number of conditions per production

Program	circuit	life	waltz	hotel	spam
Instantiations/cycle	206.0	46.8	3540.2	413.0	60.5

Table 5.7: Instantiations generated per cycle for the comparative suite

Another kind of token ordering restriction, referred to as the *cross-product* effect, arises when a token arriving at a β node matches a large number of tokens in the opposite memory. This leads to the generation of a large number of successor tokens. Checking for matches in the opposite memory involves traversing a list of tokens. Successors generated early in the traversal are available for execution before those generated late in the traversal. If this occurs at a β node high in the Rete network, it can severely limit the number of tasks that are available for execution. In such cases, most of the processors spend most of their time looking for tasks. A common situation in which this occurs is the use of sequencing tuples to direct the control-flow. Since the production system computational model provides no control-flow constructs, a commonly used technique to guide the execution of a section of code is to add a guard condition which is matched by a sequencing tuple. Sequencing tuples are also referred to as *control elements* [11] and are used by most production system programs. OPS5 and most derivatives provide explicit support for it by treating the first condition specially (the MEA selection policy). Figure 5.5 shows an example of the use of sequencing tuples. In this figure, the first condition in both productions is the guard condition and ensures that the productions will be fired in sequence. To

understand why this might severely limit the number of tasks that can be executed in parallel, consider the Kete network for the production `prod-for-stage-1` in Figure 5.6 and assume that there are a large number of tuples of type `data`. When the sequencing tuple, `(stage ^name stage-1)`, is created, it matches all the `data` tuples and generates a successor token corresponding to each one. Since the generation of successors is sequential, and since task sizes are often small, only a small number of processors can be utilized for processing these tasks.

```
(p prod-for-stage-1
  (stage ^name stage-1)
  (data ^value <d>)
  (result ^value <r>)
  -->
  (modify 3 ^value (do-stage-1-operation <d> <r>)))

(p prod-for-stage-2
  (stage ^name stage-2)
  (data ^value <d>)
  (result ^value <r>)
  -->
  (modify 3 ^value (do-stage-2-operation <d> <r>)))
```

Figure 5.5: Productions with guard conditions for sequencing

Productions that implement the constraint propagation steps in `waltz` use a sequencing tuple to delay the application of constraints until all alternatives have been generated. Figure 5.7 shows one of these productions. This production implements the constraint that if one end of a line is labeled `in`, the other end must be labeled `out`. In this case, the sequencing tuple is `(stage propagate-constraints)`. For the comparative instance of `waltz`, creation of this tuple leads to the generation over 15840 successor tasks. Figure 5.8 plots processor utilization against the number of processors for the comparative instance of `waltz`. It shows that processor utilization falls off rapidly and indicates that the maximum number of processors that can be kept busy is at most 26. Since the average number of tasks per cycle for the comparative instance of `waltz`, from Table 5.5, is as high as 29454, this clearly indicates the presence of dependencies. The productions in `waltz` are about the same size – Table 5.6 shows that the average number of conditions per production is 3.08 and the standard deviation is 0.86.

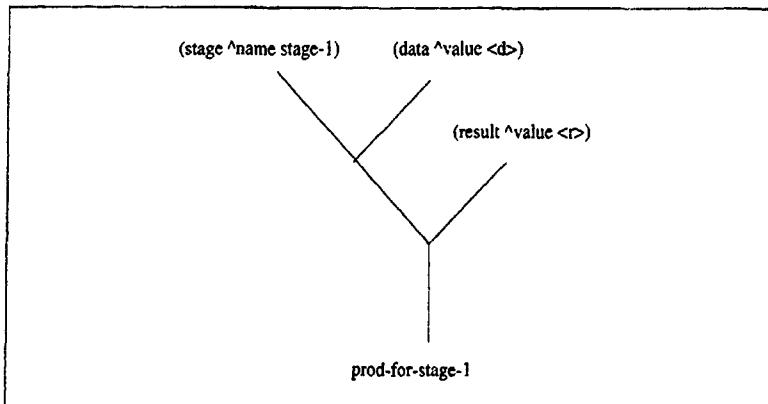


Figure 5.6: Rete network for the productions with guard conditions

Furthermore, over 3500 instantiations are generated per cycle in the comparative instance of `waltz`. Therefore, chaining dependencies are not a cause of the low number of tasks that can be executed and this limitation can be attributed, almost entirely, to the cross-product effect.

```
(parp consistent-in-out
  (stage propagate-constraints)
  (possible-line-label ^line <l> ^junction <j> ^label in ^candidate <c>)
  (labeling-candidate ^id <c> ^deleted no)
  -(possible-line-label ^line <l> ^junction <j> ^label out)
  -->
  (remove 1)
  (modify 2 ^deleted yes))
```

Figure 5.7: Constraint application production from `waltz`.

It is possible to avoid this situation by moving the guard condition to later in the production. But then, it will no longer be able to take advantage of the MEA selection policy which considers the first condition special. Furthermore, this would bring the second and third conditions to

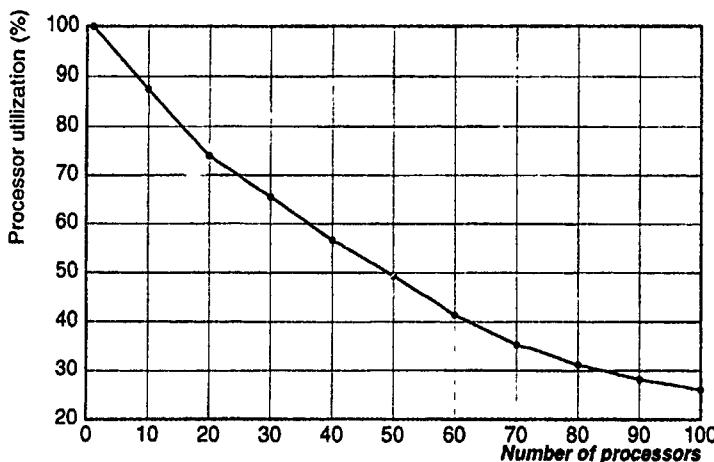


Figure 5.8: Processor utilization for the comparative instance of *waltz*.

the top of the production. There are a large number of changes made to tuples matching these conditions before the propagate-constraint stage. The incremental nature of the match algorithm will lead to matches being computed for all these changes even though no instantiations are generated. For large runs of *waltz*, this slows down the execution by over a factor of two.

Alternative network structures that attempt to reduce the chaining effect without giving up the benefits of the fixed ordering have been proposed in [113]. These structures are referred to as *constrained bilinear networks*. The basic idea is to chop up the sequence of conditions into disjoint subsequences and to connect them up as parallel branches. Since the initial conditions are often used by programmers to restrict the number of potential matches, the initial subsequence is retained as it is. Figure 5.9 shows an example.

The loss of parallelism caused by the sequential generation of successors can be alleviated if memory nodes are partitioned and a token arriving at a β node is matched in parallel with all partitions in the opposite memory. A source level approach which attempts to do this has been proposed in [93]. This approach, referred to as *copy-and-constrain*, is the production system analogue of inlining. It creates copies of the productions in which cross-products occur, each copy of a production matching a fraction of the tuples matched by the original production. To achieve a good partition, *copy-and-constrain* needs information about the sets of values that can be bound to the variables occurring in all the productions – at least the productions in which cross-products occur. Since it is not possible for the compiler to determine this, the

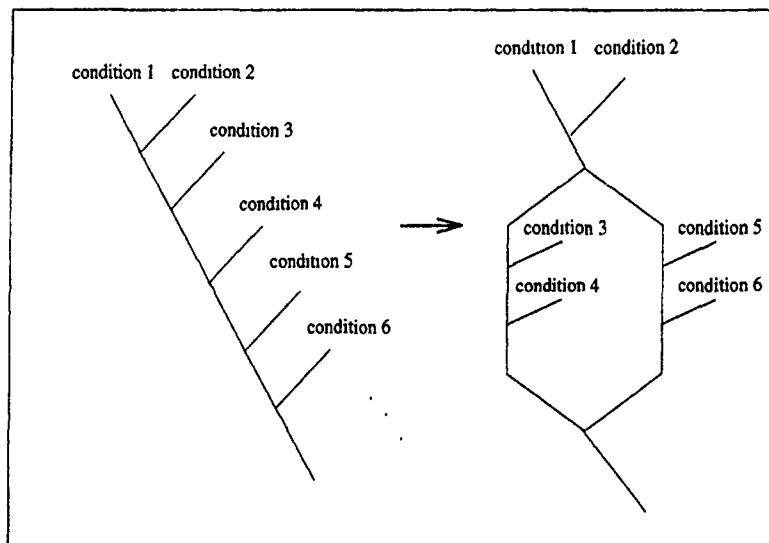


Figure 5.9: Conversion of a linear Rete network to a constrained bilinear network

copy-and-constrain optimization is usually applied manually. Another disadvantage is that repeated applications can lead to a combinatorial explosion in code size. Chapter 6 describes an alternative approach. This approach clusters the tokens in memory nodes based on the values that will be tested at the corresponding β node. A token arriving at the β node from the opposite directions traverses only the list of clusters and not the list of all tokens. This reduces the time required to process the incoming token as well as the time required to generate the successors. This approach is especially effective for cross-products occurring due to the use of sequencing tuples since in such cases, there are no tests and the entire opposite memory reduces to a single cluster. For example, there are no tests between the first and second conditions in the productions in Figure 5.5 or in the production from *waltz* in Figure 5.7.

5.2 Growth of speedups with data set size

Since the parameter space for the experiments has two dimensions, data set size and number of processors, there is a family of speedup vs number of processors curves for every benchmark,

one curve per data set size. To understand how the speedup for a given number of processors varies with the data set size, the graphs presented in this section plot all the curves for a benchmark for configurations between one and a hundred processors. Furthermore, to facilitate comparison between the parallel and parallel-match versions, that is to illustrate the extra speedup achieved with the parallel constructs, similar curves for the latter are also plotted on the same graph.

To evaluate the scalability of the speedups with data set size, graphs plotting maximum achievable speedup vs data set size are also presented. Many speedups curves presented in this chapter do not level off up to 100 processors. Since most of the curves looked like plots of negative exponential functions, the maximum achievable speedup for each curve was estimated by fitting it, in a least mean square error sense, to the following negative exponential function:

$$\text{speedup} = \text{saturation_speedup} - K e^{-\lambda p} \quad (5.1)$$

where p is the number of processors and K and λ are constants whose value depends on the characteristics of the benchmark and the data set.

5.2.1 Growth of speedups for circuit

Figure 5.10 shows the families of speedup curves for circuit. The curves for the parallel version are well spaced and curves for larger data sets lie above those for smaller data sets. This indicates that the speedups increase with the data set size. Furthermore, these curves do not level off for 100 processors indicating that larger speedups are possible with bigger configurations. However, the spacing between curves corresponding to successively larger data sets decreases as the size of the data set increases. This suggests that the growth in speedup with data set size is likely to level off for data sets significantly larger than those used in these experiments. The graph in Figure 5.11 which plots the estimated maximum speedups obtained by fitting the curves for the parallel version to Equation 5.1 supports this conclusion. It shows that estimated maximum speedup increases rapidly with data set size for data sets smaller than 125 bits. For larger data sets, the growth in estimated maximum speedup slows down.

The curves for the parallel-match version level off early (under 5 fold speedup) and most of them lie almost on top of each other. In this case, curves for larger data sets lie below those for smaller data sets. That is, as the data set size increases, the speedup decreases; the single dotted curve that appears significantly above the rest corresponds to the *smallest* data set size.

The biggest limitation on speedup in circuit is the high parallelization overhead. Figure 5.12 shows how the parallelization overhead varies with data set size. The primary cause of the parallelization overhead is the small average task size. The average task size in circuit is uniformly close to 95 instructions. Increasing the data set size by an order of magnitude (from

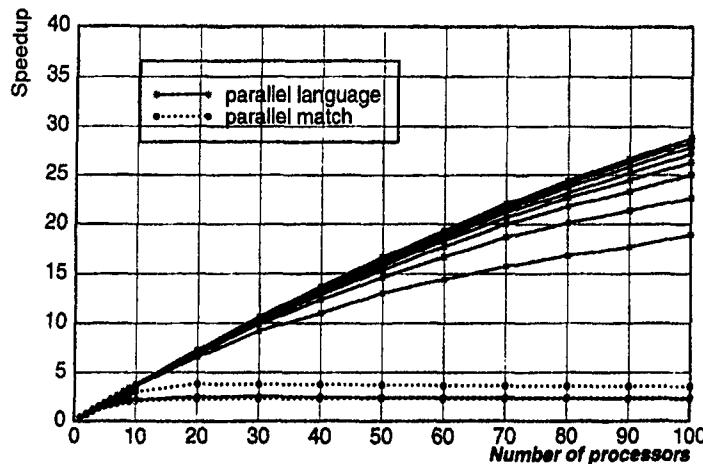


Figure 5.10: Speedup curves for circuit

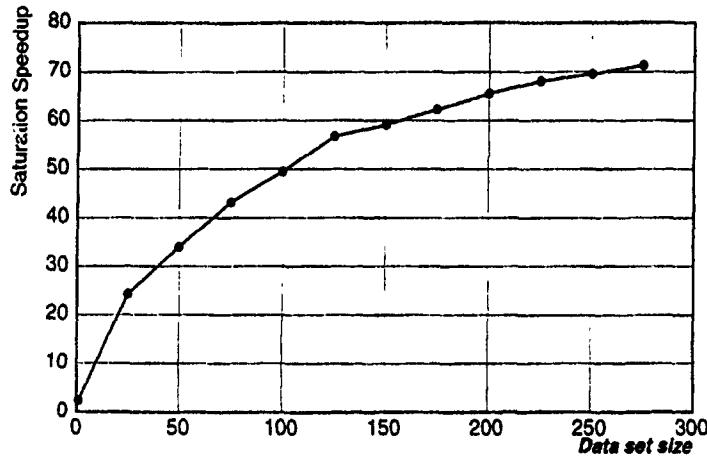


Figure 5.11: Saturation speedups for circuit

25 bits to 275 bits) does not cause much variation. Figure 5.13 plots the family of nominal speedup curves for *circuit*. It shows that up to 76 fold nominal speedup can be achieved with 100 processors. To estimate the maximum possible nominal speedup, these curves were fitted to Equation 5.1. Figure 5.14 plots the estimated maximum nominal speedup plotted against data set size.

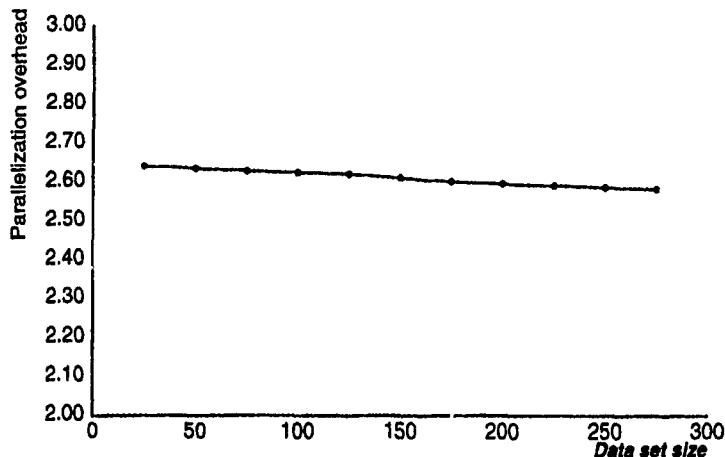


Figure 5.12: Parallelization overheads for *circuit*

All the loops in *circuit* are parallelizable. As shown in Table 5.6, there is significant variation in the lengths of the production indicating the possibility of chaining dependencies. However, the effect of chaining, if any, is masked by the large number of instantiations generated. The average number of instantiations generated per cycle grows linearly with the data set size and can be expressed as $0.49 * \text{number_of_bits} + 1.54$.

5.2.2 Growth of speedups for *life*

Figure 5.15 shows the families of speedup curves for *life*. The speedup curves for the parallel version level off and are clustered close together. This indicates that parallelism does not scale with data set size in this benchmark. An interesting point that is not apparent from the figure is that there appears to be an inflection point in the growth of speedup with the data set size. Speedups increase with data set size up to a point, after which they decrease as the data set size increases. This inflection point occurs at the data set with 60 repetitions of the

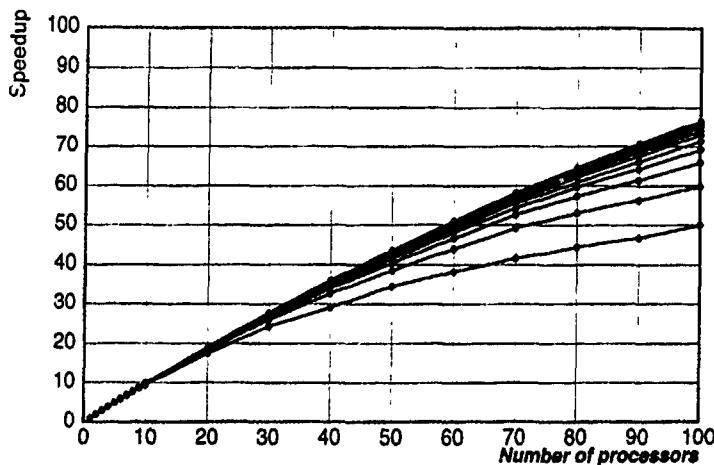


Figure 5.13: Nominal speedups for circuit

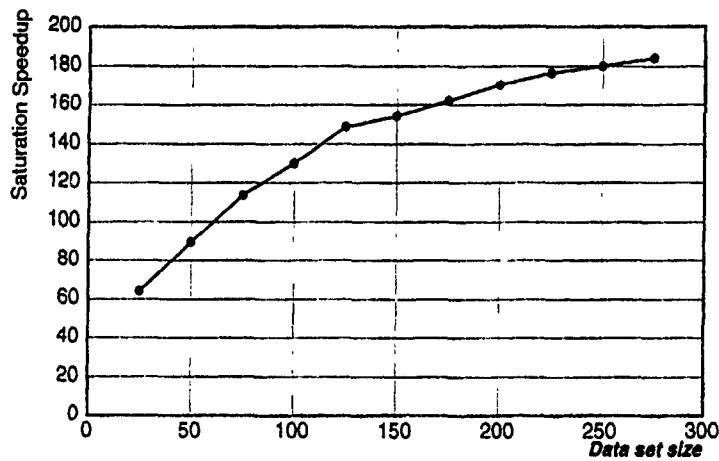


Figure 5.14: Estimated maximum nominal speedups for circuit

basic pattern.¹ In the figure, curves for all data sets lie below the curve corresponding to the 60 repetitions data set. These conclusions are supported by Figure 5.16 which shows the variation of the estimated maximum speedup computed by fitting the curves to Equation 5.1. It peaks around the data with 20 repetitions and dips thereafter. It indicates that the maximum possible speedup in *life* is a little more than 30 fold and that the maximum possible speedup for larger data sets is smaller. The discrepancy between the inflection points indicated by the two figures can be explained by examining the slopes of the curves at the right edge of Figure 5.11. Even though the curves for data sets between 20 and 60 repetitions lie progressively higher, their slope at the right edge of the graph is negative. This causes the fitting procedure (based on least mean squared error) to yield lower estimates for the maximum speedup.

The presence of the inflection point suggests that there are two competing factors, one which seeks to increase the speedup with data set size and the other which seeks to decrease it as the data set size grows. The first factor dominates for the left hand side of the graph and causes the rapid increase in the speedup, but the latter takes over as the data set size grows beyond the inflection point. Increase in the speedup as the data set size grows is caused by an increase in the number of patterns, each of which can be processed in parallel. Decrease in the speedup as the data set size grows is caused by the non-parallelizable print loop which prints the matrix. Since only two cells out of thirty in each pattern change status, the number of iterations of this loop grows much faster than the growth in the number of cells being modified each simulation cycle. While in this loop, the speedup is low; the *parallel* version can do no better than the *parallel-match* version. This implies that the inflection point in the growth of speedups with data set size may not exist for *life* data sets with different characteristics. If the fraction of mutating cells is significantly larger and the number of generations computed are significantly greater, the fraction of time spent in the sequential print loop will not be as important as it is in this benchmark. In such cases, the speedup can be expected to scale with the data set size.

Speedups in *life* are also limited by parallelization overheads and the chaining effect. The parallelization overhead for *life* is close to a factor of two. The presence of chaining is indicated by the large variation in the production length (Table 5.6) and the small number of instantiations generated per cycle (Table 5.7).

5.2.3 Growth of speedups for *waltz*

Figure 5.17 shows the families of speedup curves for *waltz*. All the curves for the *parallel-match* version lie almost on top of each other and level off around 1.75 fold speedup. The curves for the *parallel* version too level off; however, they level off at significantly higher speedups (between 6.9 and 17.5 fold). Speedups for *waltz* are the lowest in the benchmark suite. An interesting point to note is that the curves for the *parallel* version are spread out. This

¹The basic pattern for the *life* dataset is shown in Figure 4.2.

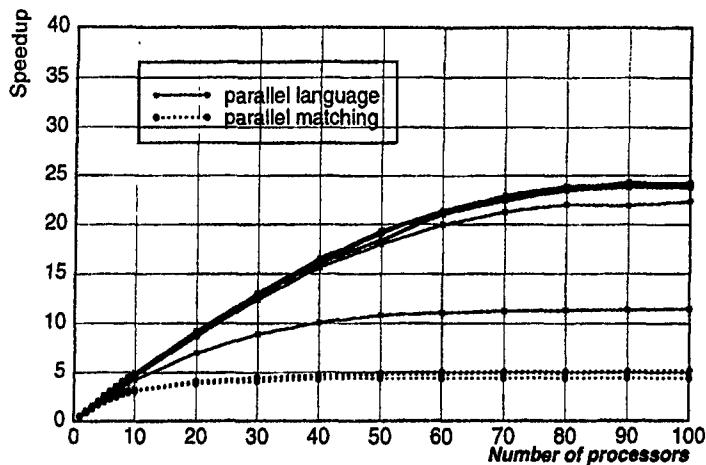


Figure 5.15: Speedup curves for life

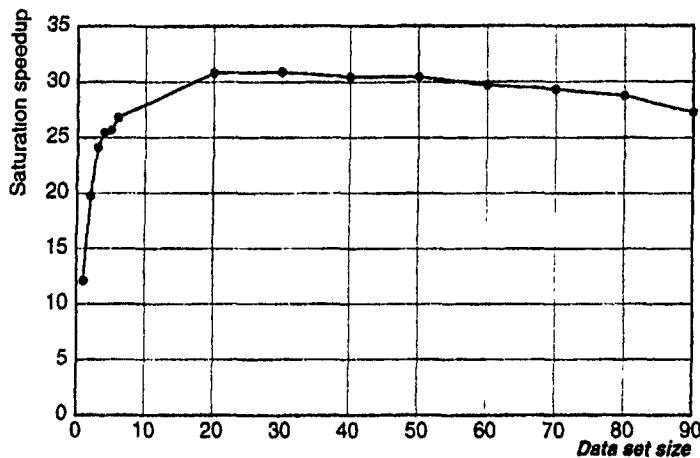


Figure 5.16: Saturation speedups for life

indicates that even though the growth in speedup with data set size is slow, it is not saturating. This is borne out by the graph in Figure 5.18 which plots the estimated maximum speedups obtained by fitting the curves for the parallel version to Equation 5.1. This graph indicates a slow but steady growth in the estimated maximum speedup. Another interesting point to note is that for the smaller data sets, the speedup curves dip beyond 60-70 processors. This effect disappears for larger data sets. These results indicate that the speedups for *waltz* are limited by dependencies between tasks.

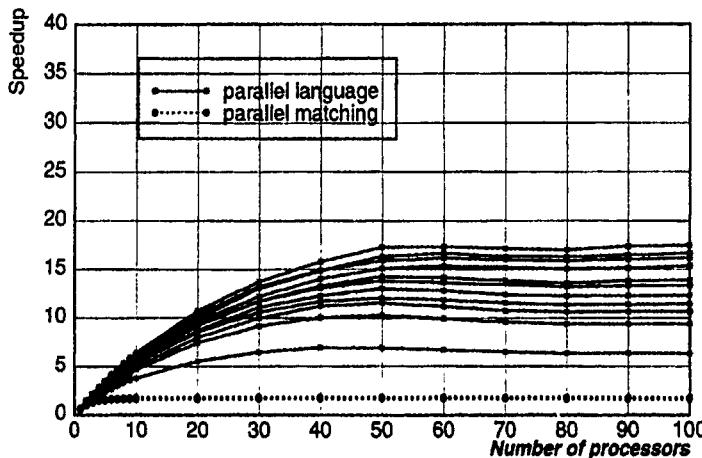


Figure 5.17: Speedup curves for *waltz*

The primary limitation on speedups in *waltz* is the cross-product effect that occurs due to the use of guard conditions to delay the application of constraints till all alternatives have been generated. This condition appears in all productions that apply constraints. Figure 5.7 shows one such production. The number of successors generated depends on the number of tuples that satisfy the second conditions in the constraint application productions. For the production in Figure 5.7, the number of tuples matching the second condition is 132 times the number of repetitions of the basic pattern. Only a few processors can be kept busy while these tasks are being executed; therefore speedups achieved in this period are low. Since the number of such tasks grows with the data set size, a consistently large fraction of the total execution time is spent in periods of low speedup. This limits the overall speedup as an Amdahl's law effect.

Another limitation on speedups in *waltz* is the parallelization overhead which is between 1.4 and 1.8 fold. The parallelization overhead reduces with the growth in data set size. This can

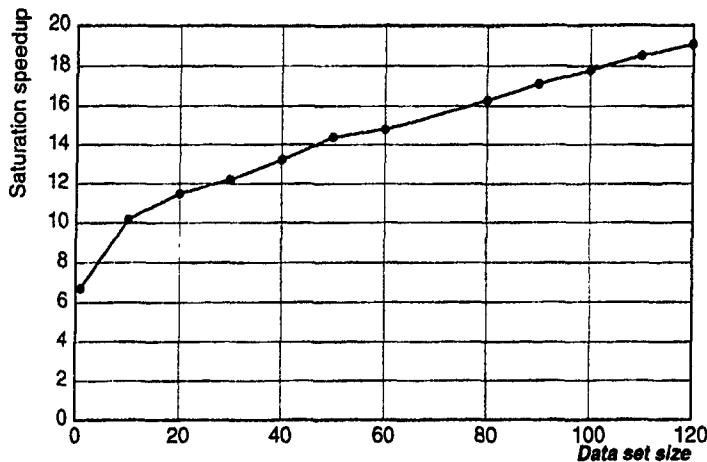


Figure 5.18: Saturation speedups for waltz

be attributed to the growth in average task size from 172 instructions to 440 instructions.

Execution of waltz consists of a sequence of constraint propagation steps. As mentioned in Section 4.1.3, due to interference between the constraints, constraints of different types have to be applied in sequence. However, all instances of the same constraint can be applied in parallel. In effect, the computation consists of a doubly-nested loop of which the outer loop, which iterates over the set of constraints is not parallelizable but the inner loop which applies the constraints is. Since, there are few constraints, each with a large number of applications, the sequential nature of the outer loop does not limit the speedups.

5.2.4 Growth of speedups for hotel

Figure 5.19 shows the families of speedup curves for hotel1. This benchmark is unique in that the parallel-match version achieves large speedups. Unlike all the other benchmarks, the two families of curves overlap all the way to 100 processors. For all other benchmarks, the speedup curves for the parallel-match version lie much below those for the parallel version. However, speedup curves for the two versions for the same data set are still widely separated. The parallel version achieves between 1.3 and 3.6 times more speedup than the parallel-match version for the same data set. Both families of curves are widely spread out indicating that the speedup grows rapidly with the growth in data set size. This is borne out

by the graph in Figure 5.20 which plots the estimated maximum speedups obtained by fitting the curves for the parallel version to Equation 5.1. This graph is close to linear and has a steep slope which indicates a steady and rapid growth in the estimated maximum speedup.

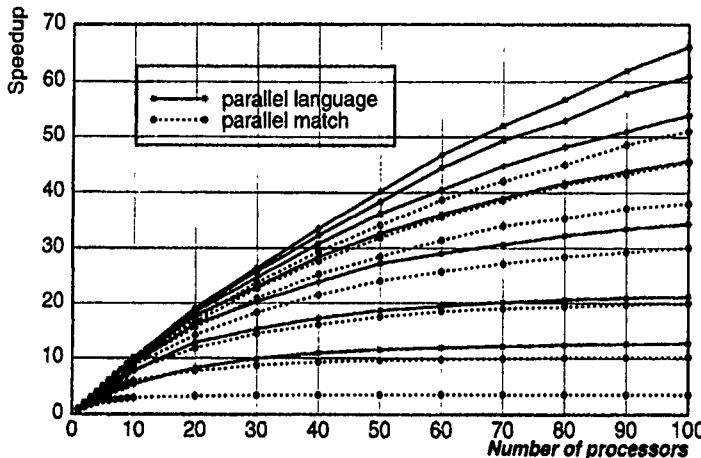


Figure 5.19: Speedup curves for hotel1

The rapid growth of speedup with data set size is achieved in spite of several limitations. Figure 5.21 shows how the average number of instantiations fired per cycle varies with the data set size. It shows that the growth quickly levels off around 27 instantiations per cycle and gradually *decreases* thereafter. This is due to the presence of several non-parallelizable loops – for example, collecting and doing the laundry, setting the tables for the banquet, cleaning restrooms.

Another limitation on speedup in hotel1 is the presence of chaining. There is a large variance in the size of productions in hotel1 – the average number of conditions per production is 3.08 with a standard deviation of 2.7 (Table 5.6).

The three major factors that allow hotel1 to achieve large speedups in spite of these limitations are the large number of instantiations generated per cycle, the rapid growth in the average task size and the near-linear growth in the number of tasks per cycle.

Even though the number of instantiations fired per cycle are small, large number of instantiations are generated per cycle. The number of instantiations generated per cycle grows from 260 to 500 for the data sets used in the experiments. Most of these instantiations are deleted from

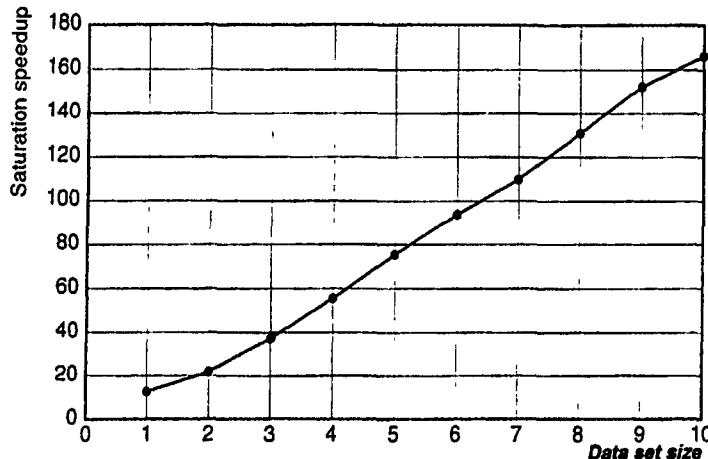


Figure 5.20: Saturation speedups for hotel

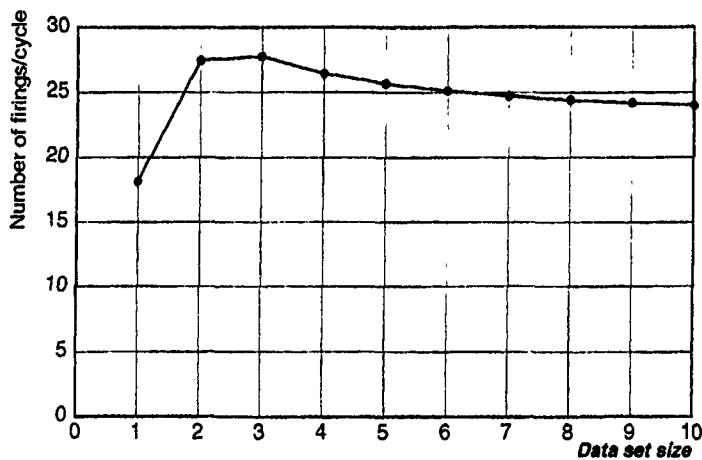


Figure 5.21: Number of instantiations fired per cycle for hotel

the conflict set without being fired. This occurs because of loops that sum a sequence of values. There are several such loops in *hotel*, for example the loop to collect the dirty linen, the loop to do the laundry and the loop to serve banquets. An example of such a loop is shown in Figure 5.22. This production sums a collection of numbers. In every cycle, an instantiation is created for every data item that has not yet been added, one of these instantiation is selected and fired. This modifies the tuple containing the accumulated value which leads to the deletion of the other instantiations. In the next *msa* cycle, this process repeats with the new accumulated-value tuple and the remaining data items. As a result, $O(n^2)$ instantiations are created, of which only n instantiations are fired. The large number of instantiations generated every cycle masks the effect of chaining – since chaining sequentializes the match tasks only within a single instantiation.

```
(p accumulate-sum
  (data-item ^value <v> )
  (accumulated-sum ^current-value <accum>)
  -->
  (modify 2 ^current-value (<accum> + <v>))
  (remove 1))
```

Figure 5.22: Example of an accumulation loop

Figure 5.23 shows how the number of tasks per cycle varies with data set size. The near-linear growth in the number of instantiations generated per cycle is the primary cause of the similar growth in the number of tasks per cycle.

Figure 5.24 shows the growth of average task size with data set size. Fitting a quadratic equation, in a least mean square error sense, yielded $0.77x^2 - 2.29x + 3.98$. This equation has been plotted as the dashed line in Figure 5.24. This growth in average task size is almost entirely due to the growth in size of match tasks. For the data sets used in the experiments, average size of action tasks remains close to 1200 instructions; average size of select tasks grows from 657 instructions to 1414 instructions; but average size of match tasks grows from 245 instructions to 32221 instructions. Since PPL uses global hashtables with overflow chaining to implement the memory nodes of the Rete network, and since each match task primarily consists of a traversal of a hashtable bucket, the quadratic growth in the task size implies a quadratic growth in the size of hashtable buckets which in turn implies a failure of the hash function to spread the tokens evenly. The hash function used in PPL (and in all other hashtable based implementations) xors the identifier of the β node to which the memory node is attached and the value(s) that will be tested at this node. This results in tokens destined for the same node and with the same

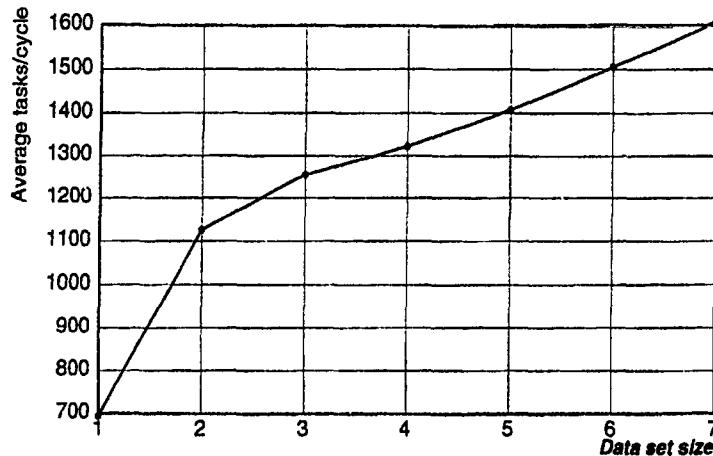


Figure 5.23: Growth of average tasks/cycle in hotel.

values being tested to be hashed to the same bucket. The most common situation in which hash function fails arises when a large number of tokens with the same tested value are destined for same β node. This is referred to as the cross-product effect and it limits parallelism in several ways, one of which is described in the previous subsection. This does not occur in *hotel* since all β nodes that test more than one token in the course of the execution have highly restrictive tests. For example, all conditions of productions that operate on rooms test the room number. The other situation in which the hash function fails arises when the node identifiers (which are integers) and value(s) being tested form sequences in the same numerical range. For example, assume the node identifiers as well as the values being tested are in the range 0 to n . In that case, $O(n)$ tokens are hashed to the each bucket since there are n combinations of numbers that xor to each value.² In *hotel*, the value most frequently tested is the room number which forms an integer sequence. The node identifiers too are sequentially allocated. This results in a linear growth in the number of tokens being hashed to the each bucket. Since the processing of each token consists of a linear traversal of a hash bucket, the average task size grows quadratically with data set size.

These effects also explain the surprising speedups achieved by the *parallel-match* version. Since a large number of instantiations are generated and deleted every cycle, there are a large

²There are two input bit combinations for any value of a result bit. Since there are $\log n$ result bits, n number-pairs xor to each number

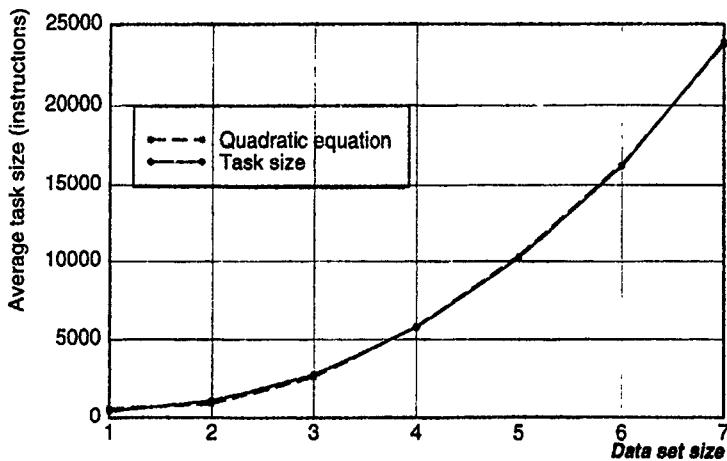


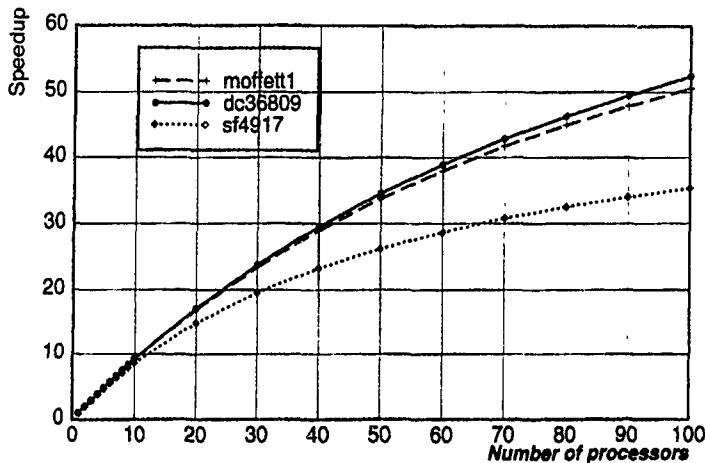
Figure 5.24: Growth of average task size in hotel.

number of tasks available even though only one instantiation is fired. Furthermore, as the size of the tasks grows (due to the failure of the hash function), the parallelization overhead becomes insignificant. This result is at variance from the conclusions of previous research which indicates large speedups are not possible for automatically parallelized production system programs.

5.2.5 Growth of speedups for spam

Figure 5.25 shows the speedup curves for the parallel version of *spam*. As mentioned earlier, *spam* spends less than 2% of its time in the match phase and the speedup achieved by the parallel-match version is expected to be less than 1.02 fold. Virtually all the speedups in this benchmark can be attributed to explicit specification of parallelism. The data set for *spam* is not parametric, that is, it can not be scaled by adjusting the value of a parameter. Table 5.8 contains the sizes of the three data sets used in the experiments. It indicates that *dc36809* is larger than *moffett1* and accordingly, its speedup curve, in Figure 5.25 lies above that of *moffett1*. However, *sf4917* is significantly larger than the other two and yet its speedup curve is significantly lower than both the other curves. The estimated maximum speedups computed by fitting the curves to Equation 5.1 are shown in Table 5.9 and support similar conclusions about the speedups in the three cases.

These results indicate that there are two competing factors, one which increases the parallelism

Figure 5.25: Speedup curves for **spam**

Data set	moffett1	dc36809	sf4917
Number of tuples	340	367	439
Number of tuple changes	57988	85781	219066
Number of tasks	343268	498281	1153869
Uniproc time (billion instrs)	15.41	23.86	41.54

Table 5.8: Size of the **spam** data sets

Data set	moffett1	dc37809	sf4917
Saturation speedup	66.93	71.25	39.19

Table 5.9: Saturation speedups for **spam**

Data set	moffett1	dc36809	sf4917
First phase	12240	17100	30060
Second phase	24387	35611	97835
Third phase (merge areas)	1020	2780	9424
Third phase (link condense)	451	621	611

Table 5.10: Number of iterations of *spam* loops

available (by increasing the time spent in highly parallel code) and the other which limits the parallelism available (by increasing time spent in sequential code or code with limited parallelism). Recall from Section 4.1.5 that *spam* has three phases. The first two phases consist entirely of parallelizable loops whereas the third phase consists of two parallelizable loops, which select seed regions and generate links between seeds and surrounding regions, and two non-parallelizable loops, which merge multiple parallel links between regions and merge overlapping functional area candidates. The number of iterations and the time spent in individual iterations depends on the characteristics of the images being analyzed. Table 5.10 shows the number of iterations of the different loops for all three data sets.

5.3 Conclusions and observations

5.3.1 Validation of hypotheses

The simulation results presented in this chapter indicate that there is no program-independent bound on the speedups that can be achieved by parallel production system programs. The benchmarks used in this investigation achieve up to 76 fold speedup with 100 processors and up to 115 fold speedup with 200 processors. As shown in the previous section, the actual speedup achieved by each benchmark depends on its own characteristics.

The detailed results in Section 5.2 show that these numbers are not an upper bound on the speedup that can be achieved by parallel production system programs. The analysis in Section 5.1.1 identifies small task size, non-parallelizable loops and dependencies between match tasks as the primary limitations on speedups in parallel production system programs.

With the exception of *life*, the speedup achieved with a given configuration grew with the data set size, the rate of growth depending on program characteristics. For *life* too, speedup can be expected to grow with data set size for data sets with larger fractions of mutating cells and larger number of generations computed.

In all cases, the speedups achieved by the parallel versions of the benchmarks were much larger than the speedups achieved by the parallel-match versions. This was

particularly so for *spam*, which spends over 98% of its time in calls to C functions and can expect less than 1.02 fold speedup from a parallel implementation of the match phase. Quantitatively, the parallel versions achieved between 1.3 and 12.5 times more speedups than parallel-match versions.³

5.3.2 Aggregate updates faster with parallel constructs

Only one instantiation can be fired at a time in sequential production system languages. Since firing an instantiation can modify only the tuples contained in the instantiation, it is not possible to atomically update large unbounded data structures, like the grid of cells for *life* or the set of lines for *circuit*, in a single cycle. Therefore, such updates have to be performed as loops spread over multiple cycles. To keep track of the updating process and to ensure atomicity of the updates, explicit flag fields have to be added to the tuples being updated. Figure 5.26 shows a production from the sequential version of *circuit* which simulates the action of an and-gate when both its input lines are on. This production achieves atomic updates by not modifying line tuples generated in the previous hardware cycle. Instead, it copies the tuples that need to be modified and updates these copies. At the end of each aggregate update, the old copies are deleted and the new copies are installed in their place (by resetting the value of the modified field to no).

This scheme modifies tuples twice for every update – one to update the value field and set the flag and the second time to reset the flag. Parallel constructs make it possible to fire an unbounded number of instantiations, and update an unbounded number of tuples in a single cycle. Therefore, it is possible to atomically update unbounded aggregate data structures in a single cycle. This eliminates the need for flag fields and tuple modifications caused by resetting of these fields. This version modifies each tuple in the aggregate only once. Figure 5.27 shows the parallel version of *simulate-and-gate-turn-on*.

This reduction in computation is made possible by taking advantage of the atomicity of the action phase – all instantiations are generated from the same tuple-space and are fired (at least logically) in parallel. Since most programs that process large amounts of data can be expected to update aggregates whose size is not known at compile time, this suggests that parallel constructs are desirable even for production system languages designed exclusively for implementation on stock uniprocessor machines.

5.3.3 Recency unsuitable for parallel languages

The algorithms used in the select phase of many production system languages[11, 30, 26] use the age of tuples, as indicated by their timetags, to order the instantiations. Instantiations

³Not considering *spam* whose parallel version shows over 50 times more speedup

```
(p simulate-and-gate-turn-on
  (and-gate ^input1 <in1> ^input2 <in2> ^output <out>)
  (line ^id <in1> ^value 1 ^modified no)
  (line ^id <in2> ^value 1 ^modified no)
  (line ^id <out> ^value 0 ^modified no)
  -->
  (copy 4 ^value 1 ^modified yes))

(p delete-old-line-copies
  (stage merge-lines)
  (line ^id <line> ^modified no)
  (line ^id <line> ^modified yes)
  -->
  (remove 1)
  (modify 2 ^modified no))
```

Figure 5.26: Atomic update of an unbounded aggregate (sequential).

```
(parp simulate-and-gate-turn-on
  (and-gate ^input1 <in1> ^input2 <in2> ^output <out>)
  (line ^id <in1> ^value 1)
  (line ^id <in2> ^value 1)
  (line ^id <out> ^value 0)
  -->
  (modify 4 ^value 1))
```

Figure 5.27: Atomic update of an unbounded aggregate (parallel).

containing recently created tuples are placed above instantiations containing older tuples. This results in tuples being processed in a *last-in-first-out* fashion. As long as only one instantiation is fired per *msa* cycle and the actions in the firing are executed sequentially, the process of timetag generation is deterministic. In such cases, recency is a stable ordering criterion since the assignment of timetags is deterministic. Parallel production system languages permit multiple instantiations to be fired in parallel. The assignment of timetags to the tuples created by parallel firings is non-deterministic and depends on the number of processors available and their relative speeds.

The use of recency as an ordering criterion in the selection algorithm for a parallel production system language introduces three problems. First, it makes it impossible to compute the control-flow graph (or even a usable approximation) at compile time since the order in which instantiations are fired can depend on run-time data values as well as the relative speeds of the processors. Second, it is likely to lead to contention for the timetag counter causing potential serialization of tuple-space updates and thereby of the match process. Third, it makes program execution sensitive to relative processor speeds which introduces subtle race conditions. For example, consider the productions in Figure 5.28. In a sequential language, it is possible to ensure that all *request* tuples are created after *operation* tuples. This ensures that in case of a conflict between the two productions, the second production will be selected for execution. That is, requests for authorization operations are given priority over requests for operations that need authorization. To achieve the same effect in a parallel language, it is necessary to use additional conditions. Depending on recency alone can result in the first production firing and performing an operation that needs authorization even though there exists a request to delete the authorization.

The rationale for the recency criterion is historical. For a long time, production system programs were used exclusively in programs modelling human cognition and reasoning. In such programs, the recency criterion is necessary to ensure responsiveness to changes in the environment[73]. It is possible to achieve responsiveness in the absence of recency. Figure 5.29 shows how an additional condition can be used to do this for the productions in Figure 5.28. In general, responsiveness can be ensured by adding an explicit timetag field and testing for it in the productions. This scheme has the advantage of allowing the programmer to create and use timetags as and when she needs responsiveness and does not force her to pay the cost of recency for programs or sections of programs that do not require responsiveness.

Contemporary programs that attempt to model human cognition do not use recency. For example, Soar[66] performs no selection and fires all matched, unrefracted instantiations in every *msa* cycle. It uses a separate decision procedure to resolve conflicts that might arise due to such unconstrained firings.

Elimination of recency also speeds up the select phase since it will eliminate the need to sort and compare timetags. In the presence of recency, determining the order between a pair of instantiations is an expensive operation since it may be (and often is) necessary to compare the

```
(p perform-authorized-operation
  (request ^operation <op> ^user <uid>)
  (authorization ^user <uid>)
  -->
  (remove 1)
  (perform-op <op> <uid>))

(p delete-authorization
  (request ^operation delete-authorization ^user <uid>)
  (authorization ^user <uid>)
  -->
  (remove 1 2))
```

Figure 5.28: Productions illustrating race conditions due to recency.

```
(p perform-authorized-operation
  (request ^operation <op> ^user <uid>)
  (authorization ^user <uid>)
  -(request ^operation delete-authorization)
  -->
  (remove 1)
  (perform-op <op> <uid>))

(p delete-authorization
  (request ^operation delete-authorization ^user <uid>)
  (authorization ^user <uid>)
  -->
  (remove 1 2))
```

Figure 5.29: Productions illustrating use of additional conditions to ensure responsiveness.

timetags for all the tuples in the instantiations. This would significantly speed up programs like `waltz` that make no use of recency but spend up to 50% of their time in ordering instantiations.

5.3.4 Multiple copies of parallel productions are desirable

Parallel productions are used for implementing loops. For loops with a large number of iterations, that is loops that process large data sets, a large number of instantiations are generated. This stresses the state-maintenance algorithms. For example, consider the production in Figure 5.30. This production simulates the operation of an and-gate. In a circuit with a large number of and-gates, the memory nodes and pnode corresponding to this production will be stressed by the generation of a large number of tokens and instantiations. In such cases, it is advantageous to make multiple copies of the parallel production in question, each copy executing a fraction of the iterations. Figure 5.31 shows how this can be done for the production `simulate-and-gate`. This is the production system analogue of loop unrolling and is an instance of the general inlining technique called *copy-and-constrain*. This technique is not specific to parallel productions and can be used for any production for which a large number of instantiations are expected. It has been successfully used in both sequential and parallel versions of all the benchmarks.

```
(parp simulate-and-gate-on
  (and-gate ^input1 <in1> ^input2 <in2> ^output <out>)
  (line ^id <in1> ^value <v1>)
  (line ^id <in2> ^value <v2>)
  (line ^id <out>)
  -->
  (modify 4 ^value (boolean-and <v1> <v2>)))
```

Figure 5.30: Production to simulate an and-gate.

As with other inlining schemes, this scheme should be used selectively. Aggressive copying can blow up the source code to several times its original size. Anecdotal evidence is provided by the original version of `hotel` in which aggressive copying had increased the code size by over ten fold.

In an effort to provide a polynomial complexity bound for the match phase, Tambe[112] has suggested imposing restrictions on the contents of the tuple-space such that each production can have at most one instantiation at a time. This is an extreme form of the copying transformation discussed above.

```
(parp simulate-and-gate-on
  (and-gate ^input1 <in1> ^input2 <in2> ^output <out>)
  (line ^id <in1> ^value 1)
  (line ^id <in2> ^value 1)
  (line ^id <out> ^value 0)
  -->
  (modify 4 ^value 1))

(parp simulate-and-gate-off-1
  (and-gate ^input1 <in1> ^input2 <in2> ^output <out>)
  (line ^id <in1> ^value 0)
  (line ^id <in2> ^value 0)
  (line ^id <out> ^value 1)
  -->
  (modify 4 ^value 0))

(parp simulate-and-gate-off-2
  (and-gate ^input1 <in1> ^input2 <in2> ^output <out>)
  (line ^id <in1> ^value 0)
  (line ^id <in2> ^value 1)
  (line ^id <out> ^value 0)
  -->
  (modify 4 ^value 0))

(parp simulate-and-gate-off-3
  (and-gate ^input1 <in1> ^input2 <in2> ^output <out>)
  (line ^id <in1> ^value 1)
  (line ^id <in2> ^value 0)
  (line ^id <out> ^value 0)
  -->
  (modify 4 ^value 0))
```

Figure 5.31: Productions to simulate an and-gate.

5.3.5 Guidelines for programming parallel production system languages

The modifications to the benchmark programs were logged to help discover guidelines for parallelizing existing production system programs and idioms for programming parallel production system languages with the hope that these would be of utility to future programmers of such languages. Most of the transformations suggested in these guidelines also improve performance on uniprocessors.

Eliminate dependence on recency: As mentioned in a previous subsection, recency makes program execution dependent on the number and relative speeds of processors. Dependence on recency for scheduling can be eliminated by using additional conditions (as shown in Figure 5.29) or explicit timetag fields.

Eliminate the use of flags for parallelizable loops: As mentioned in a previous subsection, loops in sequential production system languages need flag fields to keep track of the iterations. For loops that can be parallelized, the flag fields are redundant as all the iterations of the loop are performed in a single *msa* cycle. Figure 5.32 shows how flag fields can be eliminated for such loops. Elimination of flag fields usually improves uniprocessor performance as well since it reduces the number of modifications to the tuple-space.

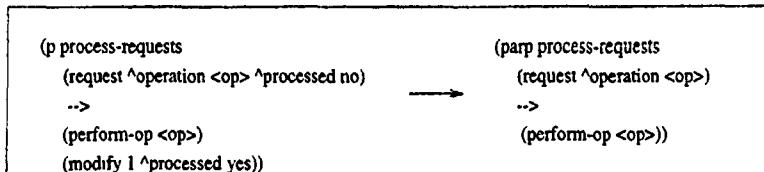


Figure 5.32: Elimination of flag fields in parallelizable loops

Eliminate dependence on sequential semantics: Sequential production system programs often depend on the total order imposed by the selection algorithm to choose one item from a set. Figure 5.33 shows an example. Assume that at any given time, there can be at most two requests for every operation. For every such pair, two instantiations will be generated – one for each permutation of the pair. The selection algorithm chooses one of these instantiations. Firing this instantiation modifies the tuples corresponding to the pair of requests which leads to the deletion of the other instantiation. These instantiations can not be fired in parallel as they would lead to multiple modifications of the request tuples as well as multiple invocation of the operations. Such programs can be parallelized by adding additional constraints which allow only one of the set of interfering instantiations to be generated. A common way is to assign numerical identifiers to the tuples being processed and use the ordering between the identifiers to restrict the generation of instantiations. Figure 5.34 shows how this can be done for the

example. Instantiations of the parallel production in this figure can process all request-pairs in parallel.

```
(p process-request-pair
  (request ^id <req1> ^operation <op> ^result nil)
  (request ^id <> <req1> ^operation <op> ^result nil)
  ->
  (modify 1 ^result (perform-op <op>))
  (modify 2 ^result (perform-op <op>)))
```

Figure 5.33: Use of total order to choose one item from a set.

```
(parp process-request-pairs
  (request ^id <req1> ^operation <op> ^result nil)
  (request ^id <> <req1> ^operation <op> ^result nil)
  ->
  (modify 1 ^result (perform-op <op>))
  (modify 2 ^result (perform-op <op>)))
```

Figure 5.34: Use of numerical identifiers to eliminate interfering instantiations.

Partition tuples with independent substructures: The intuitive way of implementing a record-like structure in the flat tuple-space is to use a single tuple for every record. However, in many cases, different fields of this tuple are (or can be) modified by separate productions. In such cases, using a single tuple for the entire record forces all its updates to be serialized. Figure 5.35 shows an example. In this example, both the new position and the new magnification of a **visual-object** are independent. However, it is not possible to compute them in parallel as the corresponding instantiation firings would attempt to modify the same tuple in parallel. This limitation can be eliminated by partitioning the record into substructures that can be independently modified and using one tuple for each such substructure. A link field is added to all these tuples; tuples for a single record have the same value for this field. Figure 5.36 shows how this can be done for the example program. In this figure, the **visual-object** field is used to link the tuples belonging to the same record.

Partitioning tuples is the production system analogue of fine-grain locking schemes used to enhance concurrency in conventional parallel languages.

```

(p compute-new-position
  (visual-object ^x <x> ^y <y>)
  (translation ^x <x-delta> ^y <y-delta>)
  -->
  (remove 2)
  (modify 1 ^x (<x> + <x-delta>) ^y (<y> + <y-delta>)))

(p compute-new-magnification
  (visual-object ^magnification <mag>)
  (zoom ^magnitude <zoom-factor>)
  -->
  (remove 2)
  (modify 1 ^magnification (<mag> * <zoom-factor>)))

```

Figure 5.35: Sequential example using a single tuple for the entire data structure

Partitioning tuples has traditionally been considered undesirable since the structure of the record is now implicit in the values of the link fields and has to be recovered every time any of the data fields have to be accessed or updated. However, partitioning can also improve the performance of sequential production system programs since modification of partitioned tuples needs to check fewer productions. For example, each tuple modification shown in Figure 5.36 needs to be tested against only one production whereas each tuple modification in Figure 5.35 has to be tested against both productions. Partitioning achieved over three fold speedup for the sequential version of hotel.

Independent concerns about representational flexibility and learning have lead to an extreme version of partitioning in Soar[66]. Soar tuples encode attribute-value pairs and have exactly three fields, two fields to contain the attribute name and the value and one field for the link value. A record with n fields is implemented in Soar by n tuples, each tuples corresponding to one field.

Move sequential loops to procedural languages: As mentioned in previous sections, sequential loops are inefficient in production system languages, in particular sequential loops that apply some function to a sequence of values and accumulate the results. It is preferable to pass the sequence to a procedural language which does not incur the unnecessary cost of matching in every iteration. Figure 5.22 shows a sequential loop in PPL that sums a sequence of values. This operation requires the generation of $O(n^2)$ instantiations of which only $O(n)$ are actually

```

{pset compute-position
  (parp compute-new-position
    (position ^visual-object <obj> ^x <x> ^y <y>)
    (translation ^x <x-delta> ^y <y-delta>)
    -->
    (remove 2)
    (modify 1 ^x (<x> + <x-delta>) ^y (<y> + <y-delta>)))

{pset compute-magnification
  (parp compute-new-magnification
    (magnification ^visual-object <obj> ^value <mag>)
    (zoom ^magnitude <zoom-factor>)
    -->
    (remove 2)
    (modify 1 ^value (<mag> * <zoom-factor>)))

```

Figure 5.36: Parallel example using partitioned tuples.

fired. Figure 5.37 shows how this loop can be moved to a procedural language (in this case, C). In this case, only $O(n)$ instantiations are generated and all of them are fired. The first production passes the sequence of values, one at a time, to C. These values are held in a data structure (`seq`) till the second production invokes the actual summation routine.

Avoid explicit sequencing tuples: Since most production system languages provide no control-flow constructs, it is tempting to use explicit sequencing tuples to schedule computation. OPS5 and most of its derivatives provide explicit support for sequencing tuples through the MEA selection strategy which considers the first condition in a production as special. As a result, sequencing tuples appear in most production system programs. Usually, such tuples contain no data and there are no consistency checks (β tests) between conditions that test them and the rest of the conditions in the production. If the second condition is matched by a large number of tuples, a large cross-product is generated which severely limits the number of tasks that can be executed in parallel at any given time.

There is no general way of eliminating the use of sequencing tuples. One technique that can be used often is to use negated conditions to delay matching of a production till some tuple has been generated. For example, the production that invokes the summing operation in Figure 5.37 uses the condition that tests for the absence of `data-item` tuples to delay the summing operation

```

(parallel pass-sequence
  (data-item ^value <v>)
  -->
  (remove 1)
  (add_item_to_sequence <v>))

(p sum-sequence
  other conditions
  -(data-item)
  -->
  (make accumulated-sum ^value (sum_c_sequence)))

sequence seq = NULL;
void add_item_to_sequence(item)
data item;
{ seq = cons(item,seq), }

int sum_c_sequence()
{ int sum = 0,
  for (; !empty(seq); seq = next(seq)) sum += value(seq);
  return(sum);
}

```

Figure 5.37: Production system and C code to sum a sequence.

till all items in the sequence have been passed to C. Another technique that can be used for many programs, especially those that perform several operations on each data item is to add a sequencing field to tuples corresponding to the data item. The productions that perform each of the operations are modified to suitably update this field. An example of this technique can be found in Figure 3.8 which illustrates pipelining of operations in production system programs.

Copy parallel productions: As mentioned in a previous subsection, generation of a large number of instantiations can stress the state-maintenance algorithms. Since parallel productions implement loops, they are likely to have a large number of instantiations. Therefore, it is desirable to make copies of parallel productions, each copy processing a part of the iterations in the loop. For an example of this transformation, see Figure 5.31 or the code for the benchmarks

in Appendix E.

5.3.6 Collection-oriented semantics essential for scalable parallelism

Semantics of existing production system languages (including PPL) dictate that each instantiation of a production must contain exactly one matching tuple for every non-negated condition in the production and no matching tuples for every negated condition in the production. A variable in a production is, therefore, bound to a single value and the actions in the *then* part of a production operate on individual tuples. This shall be referred to as tuple-oriented semantics.

The analysis in the previous sections identified three major limitations on the scalability of parallelism in parallel production system programs: small average task size, non-parallelizable loops and dependencies between match tasks. Each of these is closely related to, if not directly caused by, tuple-oriented semantics. First, match algorithms are geared to generate tuple-oriented instantiations, that is, instantiations with one tuple per non-negated condition. These algorithms perform β tests (inter-condition consistency checks) on tuple-oriented tokens, that is, tokens that contain one tuple per non-negated condition in the corresponding condition prefix. In efficient programs, this operation takes between 200 and 800 instructions. Since effective parallelization of the match phase requires decomposition at the level of individual tokens, the average task size is small.

Second, tuple-oriented semantics dictates that each variable in a production is bound to a single value and that each action in the *then* part of the production operates on a single tuple. This makes it impossible to manipulate collections of tuples (or values from tuples) as aggregates. Therefore, it is not possible to move non-parallelizable loops from production system languages, where they incur the unnecessary overhead of matching, to procedural languages.

Third, the dependencies introduced by the cross-product effect, which leads to sequential generation of the successor tokens, are caused by the desire to generate tuple-oriented tokens. Since a separate tuple-oriented successor token is needed for every pair of tokens that match at the β node involved in the cross-product, it is necessary to traverse the entire the opposite memory for each arriving token.

These problems can be alleviated by adopting a collection-oriented semantics along with corresponding match algorithms. An instantiation would, then, contain a collection of tuples corresponding to every non-negated condition. Instead of containing *one* sequence of tuples that jointly satisfy the conjunction of conditions in the *if* part, such instantiations would contain *all* sequences of tuples that jointly satisfy the conditions. For example, consider the production and the tuple-space in Figure 5.38. This production has been taken from the first phase of *spam* and tests if the perimeter of the image region matching the first condition is within the acceptable range for a runway. Tuple-oriented semantics leads to the generation of a separate instantiation for every image region (as shown in the figure). On the other hand,

collection-oriented semantics would lead to the generation of a single instantiation for all image regions (as shown in the figure). In this case, the first condition is matched by the collection of the three region tuples.

```

(pset runway-test-perimeter
  (parp RTF**runway-match-perimeter
    (region ^name <name> ^perimeter <value> ^identifier <id>)
    (rtf-rule-constants ^ruleset runway-match-attributes
      ^attribute perimeter ^constants <lbound> <ubound>)
    (rtf-stage ^name match-features)
    -->
    (call spam_rtf_match_feature <id> <value> <lbound> <ubound> runway 0))
  )
Tuple space:
  1: (region ^name region1 ^perimeter 27 ^identifier 97)
  2: (region ^name region2 ^perimeter 32 ^identifier 99)
  3: (region ^name region3 ^perimeter 42 ^identifier 101)
  4: (rtf-rule-constants ^ruleset runway-match-attributes
      ^attribute perimeter ^constants 30 40)
  5: (rtf-stage ^name match-features)

Instantiations (tuple-oriented semantics):
  <1,4,5>, <2,4,5> and <3,4,5>

Instantiations (collection-oriented semantics):
  <(1,2,3),{4},{5}>

```

Figure 5.38: Instantiations for tuple-oriented and collection-oriented semantics

Since a collection-oriented token contains a collection of tuples for every condition, β tests in match algorithms that support such tokens consist of comparing collections of unbounded cardinality. This is likely to increase the average task size, thereby reducing the parallelization overhead.

In collection-oriented semantics, each variable is bound to a collection of values and actions in the then part operate on collections of tuples (or values from tuples). For example, in Figure 5.38, the variable <id> is bound to the collection {97,99,101}. The foreign function called, `spam_rtf_match_feature()`, is now passed collection-valued arguments. This

allows the programmer to move sequential loops from production system languages to procedural languages. Figure 5.39 shows how the sequential loop in Figure 5.22 that sums a sequence would be implemented in a collection-oriented production system language. Similar transformations are possible for the print loop in life as well as for the merging loops in spam.

```
(p sum-sequence
  (data-item ^value <v>)
  -->
  (remove 1)
  (make accumulated-sum ^value (sum_sequence <v>)))

int sum_sequence(seq)
sequence seq;
{
  for (; !empty(seq); seq = next(seq))
    sum+= value(seq);
  return(sum);
}
```

Figure 5.39: Summing a sequence in a collection-oriented production language.

Match algorithms based on collection-oriented semantics would also eliminate the sequential generation of successor tokens for cross-products, as it happens in waltz, by partitioning memories into collections of tokens with the same value(s) for the fields that are tested at the corresponding β node. It is, then, necessary to traverse just the list of such collections and not the list of all tuples in the memory. While, it is possible to use partitioned memories for tuple-oriented match algorithms, generation of tuple-oriented tokens is unable to avoid creating a separate token for every tuple in the memory. For cross-products caused by the use of sequencing tuples, a collection-oriented match algorithm would be especially effective since there is no test between the guard condition and its successor. A collection-oriented match algorithm would be able to add entire list of tokens in the memory node to the successor token by a single pointer assignment.

The subsequent chapters of this dissertation, 6, 7 and 8, investigate the design, implementation and performance of collection-oriented production system languages.

5.3.7 Performance on real machines

The simulation results presented in this chapter assume a uniform memory access model and use instruction counts as a measure of the execution time. They provide a measure of the amount of parallelism available that takes the scheduling costs into consideration but does not take the details of the memory system into consideration. In effect, the speedups reported by the simulator constitute approximate upper bounds on the speedups that can be achieved on a real multiprocessor. These results help establish that there is no general program-independent limit on the speedups that can be achieved by parallel production system programs. It is also important to understand what speedups can be expected on real machines.

To explore the effect of non-uniform memory access, a subset of the experiments were rerun on an Omron Luna-83K multiprocessor. This machine has four Motorola 88000 processors, 16 megabytes of main memory and 16 kilobytes data and instruction caches. Given the small cache size and the fast processors, the Omron machine is significantly distant from the model assumed by the simulator. Tables 5.11, 5.12 and 5.13 present results for *life*, *circuit* and *hotel* respectively. These results are for three processors, the remaining processor is not used in this experiments as it is used for operating system purposes. These tables report results from three versions of each program – the *baseline* version, that is the version with sequential constructs compiled for uniprocessor execution, the *parallel-model* version, that is the version with parallel constructs compiled for uniprocessor execution and the *parallel* version, that is the version with parallel constructs compiled for multiprocessor execution. The first column, labelled *par constructs*, contains the ratio of the execution times for the *parallel-model* and the *baseline* version. It is a measure of the cost of using parallel constructs (or the fraction of the speedup that can be attributed to the use of parallel constructs). The second column, labelled *par overhead*, contains the ratio of the execution time of the *parallel* version running on one processor and the *parallel-model* version. It is a measure of the cost of running the program in parallel. The parallelization costs include the costs of creating and scheduling tasks, handling shared memory, locking and handling out-of-order computation. The third column, labelled *speedup* contains the ratio of the execution time of the *parallel* version running on three processors and the *baseline* version. This is the end-to-end speedup achieved by the program on three processors. The numbers in parentheses are simulator predictions for the corresponding values. These results also allow a breakdown of overall speedup into two parts – the speedup achieved by the use of parallel constructs alone (on a uniprocessor) and the speedup achieved by exploiting the parallelism on three processors.

The breakdown of the speedup shows that *life* and *circuit* achieve a significant portion of their speedup from parallelism whereas *hotel* achieves almost all of its speedup from the use of parallel constructs and almost no speedup from parallelism. As discussed in Section 5.2.4, sequential assignment of node identifiers by the PPL implementation leads to a failure of the hash function for the token storing hashtables. This causes a large number of tokens to be

Data set	par constructs	par overhead	speedup
20	1.17 (1.11)	2.94 (2.07)	0.62 (1.47)
30	1.18 (1.11)	3.04 (2.07)	0.61 (1.49)
40	1.21 (1.12)	3.08 (2.03)	0.55 (1.51)
50	1.30 (1.13)	2.85 (2.00)	0.52 (1.53)

Table 5.11: Results for life on an Omron

Data set	par constructs	par overhead	speedup
125	0.85 (0.83)	2.61 (2.62)	0.77 (1.13)
175	0.84 (0.83)	2.56 (2.60)	0.83 (1.14)
225	0.82 (0.82)	2.55 (2.59)	0.84 (1.15)
275	0.81 (0.82)	2.52 (2.58)	0.85 (1.15)

Table 5.12: Results for circuit on an Omron

Set	par constructs	par overhead	speedup
2	0.36 (0.91)	1.19 (1.15)	3.01 (2.58)
3	0.31 (0.88)	1.06 (1.03)	3.39 (2.91)
4	0.30 (0.88)	1.03 (1.00)	3.45 (3.02)
5	0.29 (0.88)	1.02 (0.99)	3.53 (3.05)

Table 5.13: Results for hotel on an Omron

hashed to a small set of buckets which increases both the average task size and the contention for the hash buckets. Since a large number of tasks compete for the same buckets and since each of these tasks locks the bucket for a relatively long period, *hotel* is unable to achieve significant speedup from parallelism. It is easy to modify the implementation to improve the performance of the hash function (by using pseudo-random numbers as identifiers of the nodes in the Rete network). Table 5.14 presents results for *hotel* using a version of the PPL implementation that assigns random node identifiers. It shows that for a successful hash function, *hotel* achieves between 1.16 and 1.36 fold speedup from parallelism.

For all the three benchmarks, the parallelization overhead is significant, causing between 1.68 and 3.08 times slowdown. Three processors are too few to achieve significant speedup for such a fine-grained implementation. This is a familiar phenomenon for languages that exploit fine-grain parallelism. Similar results have been shown for parallel versions of Lisp/Scheme that use *futures* for specifying fine-grain parallelism [58, 63]. As shown in Table 5.3, the average task size in production system programs is often small. This is consistent with the findings of Gupta *et. al* [41] and Tambe *et. al* [113].

Data set	par constructs	par overhead	speedup
4	0.60	1.68	1.36
5	0.48	1.74	1.56
6	0.39	1.85	1.70
7	0.35	1.93	1.83
8	0.31	1.95	1.89
9	0.29	2.02	2.01

Table 5.14: Results for hotel for a modified PPL implementation

Data set	speedup (print)	speedup (no print)
125	1.75	1.75
175	1.79	1.85
225	1.85	1.98
275	1.87	2.01

Table 5.15: Results for coarse-grain decomposition of circuit

Even though, in general, effective parallelization of production system programs requires fine-grain decomposition, it is possible to partition particular programs at a coarser grain which is more suitable for small cardinality multiprocessors. For example, Harvey *et. al* decomposed the first two phases of SPAM into large tasks that could be run independently of each other and used a task queue to schedule them on an Encore Multimax[48]. They reported speedups up to 12.5 fold using 14 processors on a single Multimax and up to 15 fold using 23 processors on a pair of Multimaxes which shared virtual memory using a *netmemory server* [31]. As discussed in Section 4.1.5, the first two phases of SPAM consist of triply nested loops with no dependencies between the iterations. Table 5.15 shows results for a coarse grain decomposition of circuit. These experiments used an SPMD (single program, multiple data) model. The input data, a linear feedback shift register, was divided into three pieces. The communication between the segments is limited to the values of the boundary lines. The PPL implementation was modified to use a blackboard to communicate these values. The results indicate that, unlike the fine-grain decomposition, significant speedup can be achieved for this decomposition. The first column contains the speedups for the benchmark version of circuit. Since printing the results causes a sequentialization, another set of experiments were run with a modified version of circuit that did not print the results. The results indicate that this yields a small incremental speedup for larger data sets.

Not all programs are amenable to coarse-grain decomposition. Of the other benchmark programs, spam and life can be easily partitioned in a coarse manner whereas hotel and

Program	data set	pixie time	actual time	ratio
life	50 repetitions	33.4s	76.5s	2.29
circuit	200 bits	14.6s	36.2s	2.48
hotel	4 floors	122.0s	296.4s	2.43

Table 5.16: Comparison of estimated and actual execution times on a uniprocessor

waltz less so. In particular, waltz is unsuitable for coarse-grain decomposition as the order in which the constraints are applied determines the result and some orders can cause the program to fail to find a consistent labelling.

The primary data-structure in production system programs is the tuple-space which provides no structuring or clustering constructs and can be accessed from anywhere in the program. Therefore, in general, the locality of reference for production system programs can be expected to be low. Table 5.16 provides evidence for this by comparing the execution time estimated by the pixie program⁴ which assumes that each instruction takes one cycle with the actual end-to-end execution time on a Decstation 5000/200 with a 64 megabyte main memory and 64 kilobyte instruction and data caches. The high ratio between the actual execution time and the estimated execution time indicates that there are a lot of cache misses and that the locality of reference is low.

Collection-oriented production languages and match algorithms are one possible approach to improving speedups on non-uniform memory access machines. As mentioned in the previous section, tasks in collection-oriented match algorithms can be expected to be larger than tasks in tuple-oriented match algorithms since they compare collections of unbounded cardinality instead of individual entities. Similarly, collection-oriented actions can be expected to increase the amount of work done for firing individual instantiations. Collection-oriented match algorithms can also be expected to improve the locality of reference since they partition the set of tuples that match individual conditions into equivalence classes based on the values that are tested at the subsequent β nodes (see Section 6.2 for a description of equivalence classes in collection-oriented match algorithms). Instead of testing every single tuple in a right memory, they test only one tuple per equivalence class.

In the beginning of this section, it was mentioned that the speedups reported by the simulator are *approximate* upper bounds on the speedups that can be achieved on a real multiprocessor. They are approximate for two reasons. First, it is possible that the different order in which tokens are processed by a multiprocessor might change the amount of work to be done for processing a change to the tuple-space (see Section 4.3.1.1 for an example and further details). Second, the combined size of caches on a multiprocessor is usually much larger than the size of

⁴This program is available on Decstation 5000/200s

a cache associated with a single processor. Since poor locality of reference is a major problem for production system programs, it is possible that the larger cache size could offset some or all of the effects of the contention for the communication medium.

Chapter 6

Collection-oriented Match

The primary cause of poor scalability of production system match algorithms is the combinatorial explosion in the number of potential matches as the size of the tuple-space grows [38, 80, 116]. This combinatorial growth is caused by the need to match conjunctive rule conditions. Since every conjunct can potentially match the entire tuple-space, the number of potential matches, in the worst case, is $O(|D|^n)$ where $|D|$ is the cardinality of the tuple-space and n is the length of the longest conjunction.

For example, consider the production and the tuple-space in Figure 6.1. This production creates development teams for new projects such that each team has one hardware and one compiler expert who have worked together previously. Each condition in the production matches four tuples, the first condition matches the hardware experts, T1-4, and the second condition matches compiler experts, T5-8. Without the requirement of common previous experience, that is, without the common variable $\langle p \rangle$, this production would have had $4 \times 4 = 16$ instantiations. With the restriction, eight instantiations are generated – four with employees previously with the warp project ($\langle T1, T5 \rangle, \langle T1, T6 \rangle, \langle T2, T5 \rangle, \langle T2, T6 \rangle$) and four with employees previously with the psm project ($\langle T3, T7 \rangle, \langle T3, T8 \rangle, \langle T4, T7 \rangle, \langle T4, T8 \rangle$).

Current match algorithms are *tuple-oriented*, that is, they consider individual tuples as the primary objects to be matched and generate a separate combination of tuples for every way in which a conjunction of conditions can be matched. As the number of tuples matching individual conditions grows, the number of ways in which their conjunction can be matched grows as a combinatorial product. This chapter presents a new approach to matching which attempts to mitigate the combinatorial explosion by not generating a separate combination of tuples for every way in a condition-conjunction can be matched. First, it presents and discusses the key idea behind the approach. It then presents a new match algorithm based on this approach. The tuple-oriented nature of the existing match algorithms is closely tied to the tuple-oriented semantics of the languages they are used to implement. The next chapter discusses language semantics and programming styles supported by the new class of match algorithms introduced in

```

(p create-development-team
  (employee ^name <n1> ^previous-project <p> ^expertise hardware)
  (employee ^name <n2> ^previous-project <p> ^expertise compilers)
  -->
  (make team ^hardware-expert <n1> ^compilers-expert <n2>))

T1: (employee ^name Tom ^previous-project warp ^expertise hardware)
T2: (employee ^name Dick ^previous-project warp ^expertise hardware)
T3: (employee ^name Harry ^previous-project psm ^expertise hardware)
T4: (employee ^name John ^previous-project psm ^expertise hardware)
T5: (employee ^name Ram ^previous-project warp ^expertise compilers)
T6: (employee ^name Shyam ^previous-project warp ^expertise compilers)
T7: (employee ^name Madhu ^previous-project psm ^expertise compilers)
T8: (employee ^name Jadhu ^previous-project psm ^expertise compilers)

```

Figure 6.1: Example production and tuple-space.

this chapter. The subsequent chapter presents experimental results comparing the performance of a tuple-oriented match algorithm and its collection-oriented analogue. The work presented in these three chapters has been done jointly with Prof. Milind Tambe of the Information Sciences Institute, University of Southern California.

6.1 The key idea

In the above example, consider the four instantiations containing employees previously with the warp project. The tests in the production discriminate only between employees with different expertise – the hardware experts match the first condition and the compiler experts match the second condition. They do not discriminate between different employees with the same expertise and the same previous project. However, tuple-oriented match algorithms provide discrimination between even such employees by generating separate instantiations for them. For example, generation of $\langle T1, T5 \rangle$ and $\langle T1, T6 \rangle$ discriminates between Ram and Shyam both of whom are compiler experts previously with the warp project. Therefore, tuple-oriented algorithms pay for discrimination between all tuples even if the program does not require it.

The key idea underlying the alternative approach proposed in this chapter is that match algo-

rithms should provide only as much discrimination as required by the productions. In other words, "if you do not discriminate, you do not pay for it".¹

Tuples that satisfy exactly the same subset of tests, like the tuples corresponding to employees with the same expertise and common previous project in the above example, are *equivalent* in the context of the given program. A match approach that does not discriminate between such tuples must hold all mutually equivalent tuples as a collection. Therefore, the primary objects to be matched are collections of equivalent tuples instead of individual tuples. Such an approach is *collection-oriented* since all operations, matching, deletion, modification etc., are performed on collections of tuples. A match algorithm based on this approach matches each condition with a collection of tuples and generates *collection-oriented instantiations* which have a collection of tuples corresponding to each condition in the production. Since tuples in each collection are equivalent, all tuples in an instantiation are guaranteed to be mutually consistent. For the example in Figure 6.1, a collection-oriented match algorithm will generate two instantiations. The first instantiation, containing previous members of the warp project, is $\langle \{T1, T2\}, \{T5, T6\} \rangle$ and the second instantiation, containing previous members of the psm project, is $\langle \{T3, T4\}, \{T7, T8\} \rangle$. Tuples in each of these instantiations are mutually consistent, that is they all have the same value of the previous-project field. Collection-oriented instantiations contain exactly the same information about consistency of matching tuples as corresponding tuple-oriented instantiations, only in a terse form. The tuple-oriented instantiations corresponding to a collection-oriented instantiation can be easily generated by creating a cross product of its component collections. For example, the tuple-oriented instantiations containing previous members of the warp project can be created as a cross-product of the two collections in the corresponding collection-oriented instantiation, that is, $\{T1, T2\}$ and $\{T5, T6\}$.

Another way of looking at this is to view the matching operation as a retrieval in a relational database. The tuple types correspond to relations, the intra-condition tests (the α tests) to *selection* operations and the inter-condition tests (the β tests) as *join* operations. The example in Figure 6.1 corresponds to the self-join on the employee relation shown in Figure 6.2. From this point of view, tuple-oriented match algorithms create explicit or eager joins – where the join is completely computed at the earliest possible opportunity. On the other hand, collection-oriented match algorithms would create implicit or lazy joins – where the join formation is delayed as long as possible. Tuple-oriented match algorithms represent the result of a join as a collection of pairs whereas collection-oriented match algorithms would represent it as a pair of collections from which the join can be generated.

¹And, fairly enough, if you do discriminate, you do pay for it!

```

select emp1.name, emp2.name
from employee emp1, employee emp2
where emp1.expertise = hardware
and emp2.expertise = compilers
and emp1.previous_project = emp2.previous_project

```

Figure 6.2: SQL version of the example production

6.1.1 Analysis

At this point, two questions arise naturally. First, why should collection-oriented match algorithms be expected to scale better than tuple-oriented ones and second, when should they be expected to scale better. This subsection attempts to answer these questions analytically. Empirical results supporting the analysis in this subsection will be presented in Chapter 8.

Production system semantics require matching conjunctions of conditions, each condition being capable of matching an arbitrary subset of the tuple-space. Therefore, the number of instantiations (and tokens) will always be a combinatorial product. In tuple-oriented algorithms, the factors in this product are the number of tuples that match individual conditions whereas in collection-oriented match, combinatorial factors are the number of collections, or equivalence classes, of tuples that match individual conditions. Since the number of collections matching each condition can be expected to be much smaller than the total number of tuples, the overall combinatorial product can be expected to be significantly smaller for collection-oriented match algorithms. A reduction in the number of instantiations and tokens causes a corresponding reduction in execution time since fewer combinations have to be generated and updated.

As the number of tuples matching individual conditions grows, a combinatorial explosion in the number of instantiations and tokens is very likely for tuple-oriented match algorithms. This explosion can be avoided in collection-oriented match algorithms if the number of collections does not grow, that is, if the additional tuples for larger data sets fall in the same equivalence classes as the tuples for smaller data sets. In the best case, all the tuples matching each condition are equivalent and only one collection-oriented instantiation is generated for every production. For the production in Figure 6.1, this would occur if all employees had worked on the same previous project. For a production with m conditions, each of which match n_i , $i \in 0, \dots, m-1$, tuples, such an instantiation would take space proportional to $\sum n_i$. The corresponding tuple-oriented instantiations generate the complete cross-product of these tuples and would need space proportional to $\prod n_i$. In such a case, collection-oriented match can reduce the number of instantiations and tokens from a high-order polynomial in the size of the tuple-space (with a degree equal to the length of the longest production) to a linear function of the size of the

tuple-space. Note that the best case for collection-oriented match is the same as the worst case of tuple-oriented match. Tuple-oriented match is the degenerate case of collection-oriented match. It corresponds to the case where every collection contains at most one tuple.

Since collection-oriented match requires no restriction on the expressiveness of the productions or the contents of the tuple-space, it does not reduce the worst-case space and time complexity of the production match problem. It is still possible to encode NP-complete problems such as subgraph isomorphism within the match of a single production. For a description of how this encoding can be done, see [76].

The primary factors that govern the performance improvement achieved by collection-oriented match are the number and the average size of collections. Large improvements over tuple-oriented match can be expected for programs which generate a large number of big collections. Programs which can be expected to have such behavior are those that have tests with low selectivity/discrimination relative to the contents of the tuple-space. At one end of the selectivity spectrum are programs with very large tuple-spaces, like active databases. No matter how selective the tests are, continued growth in tuple-space size will eventually generate large collections. On the other end of the spectrum are programs with small tuple-spaces and poor selectivity. An example of this are the *expensive chunks* which occur in Soar[115]. These productions are automatically generated as a part of the learning process and are, in effect, a generalized summary of the problem-solving. The generalization process reduces selectivity by replacing constants by variables.

6.2 Collection-oriented match algorithms

The major difference between tuple-oriented and collection-oriented match algorithms is the latter group equivalent tuples into equivalence classes whereas the former do not. A tuple-oriented match algorithm can be converted to its collection-oriented analogue by adding mechanisms to create and maintain equivalence classes of tuples. Primary data structures used in tuple-oriented match algorithms are right memory nodes (containing tuples), left memory nodes (containing tokens) and the pnodes (containing instantiations). The major operations on these data structures are addition, deletion and searching for matches. The rest of this section discusses and evaluates the alternatives for modifying these data structures to deal with equivalence classes of tuples. The next section presents Collection Rete, the collection-oriented analogue of the Rete algorithm.

The following discussion assumes that every β node has its own memory nodes. This implies that every memory node has a unique β node as its destination and therefore, has a unique test associated with it. This assumption is made by a large class of match algorithms, particularly by algorithms that use hashtables to store the contents of the memory nodes and by algorithms that attempt to maximize concurrency (see Section 2.5 for details).

6.2.1 Right memory nodes

Conceptually, a right memory node in tuple-oriented match algorithms consists of a list of tuples that match the corresponding condition. This list can be easily partitioned into a list of equivalence classes based on the value of the field being tested. For equality tests, it is sufficient to find the equivalence class with the appropriate value. Additional structure on this list would be necessary for efficient implementation of relational tests. Possibilities include ordering the list of equivalence classes and arranging them in a binary search tree. If the number of equivalence classes grows large, it is possible to use a hashtable to speed up the search operations.

The above discussion assumes that the destination β node tests only one field of the tuples contained in the memory node. While, this is not *always* the case, it *almost* always is. The average number of tests per β node for the benchmarks used in Gupta's thesis [38] was between 0.37 and 1.27. The corresponding range for benchmarks used in the parallelism experiments described in previous chapters of this thesis is 0.23 to 0.74. In the relatively rare case of multiple fields being tested, any one of the fields being tested can be used to partition the memory node.

Addition of a tuple: The memory node is searched for the equivalence class that this tuple would belong to. If such an equivalence class is found, this tuple is added to it. Else, a new equivalence class is created for it.

Deletion of a tuple: The memory node is searched for the equivalence class that this tuple belongs to and the tuple is deleted from it. If this tuple is the last member of the equivalence class, the equivalence class is deleted from the memory node.

Searching for a match: This depends on the test(s) being performed at the destination β node. The common case is a single equality test. The match, in this case, consists of finding the appropriate equivalence class. There exists a match if and only if such an equivalence class exists. Inequality tests can be performed in a similar manner only with an inverted test – that is, there is a match if and only there is no appropriate equivalence class. Searching for matches for a relational test can take advantage of ordering between equivalence classes to determine the list of equivalence classes whose members satisfy the test. Checking if an equivalence class satisfies a test can be optimized by caching the value to be tested.

For β nodes that test multiple fields of a tuple, the searching can be initially done with the just one of the fields and the subsequent tests can be applied only to the tuples that satisfy the first test.

Partitioning based on values being tested reduces the average number of comparisons that have to be done for every instance of the search operation. In this, it is similar to hashing the contents of the memory nodes as has been recommended by Gupta[38]. The relative performance of these two schemes remains to be explored. Note that it is possible to build hybrid schemes that use hashing to speed up the search for equivalence classes.

6.2.2 Left memory nodes

Conceptually, a left memory node in tuple-oriented match algorithms consists of a list of tokens. Each token corresponds to a sequence of tuples that matches a condition prefix. A token consists of a sequence of slots, each of which contains a pointer to a tuple. Tokens can be easily modified to use equivalence classes of tuples. Instead of pointing to a single tuple, each slot points to a collection of tuples. Such tokens are referred to as collection-oriented tokens. In the rest of this dissertation, token should be taken to mean collection-oriented token unless mentioned otherwise.

Operations on collection-oriented tokens: There are three operations possible on collection-oriented tokens: *extension*, *merging* and *breaching*. Extension takes a token with n slots and a collection of tuples and creates a token with $n + 1$ slots with the given collection in the last slot. A token can be extended by a collection if and only if all the tuples in the collection are consistent with all the tuples already contained in the token.

Merging takes two tokens with equal number of slots and creates a new token which has the same number of slots and which contains all the tuples contained in the original tokens. A pair of tokens can be merged if and only if they differ in only one slot. The collections in all the other slots remain unchanged; the collections in the differing slot are merged. For example, $\langle (T1), (T2, T3) \rangle$ and $\langle (T1), (T5) \rangle$ can be merged to form $\langle (T1), (T2, T3, T5) \rangle$. The correctness of merging can be easily shown. Let the first token be $T.D_1$ and the second token be $T.D_2$, where D is the differing slot and T corresponds to the rest of the slots. Since, T is consistent with both D_1 and D_2 , and since there are no tests between different tuples matching the same condition, it is safe to merge them and create $T.(D_1 + D_2)$. On other hand, it is not possible to merge tokens that differ in more than one slot. To show that, let the first token be $T.D_{11}.D_{12}$ and the second token be $T.D_{21}.D_{22}$. In this case, tuples in D_{11} may or may not be consistent with tuples in D_{22} . Similarly, tuples in D_{21} may or may not be consistent with tuples in D_{12} . Merging such tokens would violate the mutual consistency requirement.

Breaching takes a token, a tuple and a test that can be applied to the two. It splits the token into two subtokens, one containing the tuples that satisfy the test for the given tuple (the *consistent* subtoken) and the other contains the tuples that do not satisfy the test (the *inconsistent* subtoken). Breaching is the inverse of merging. For example, suppose the original token is $\langle (T1), (T2, T3, T4) \rangle$ and the given test checks for consistency between the second slot in the token and the given tuple. If the test succeeds for $T2$ and $T3$ and fails for $T4$, the breaching operation generates $\langle (T1), (T2, T3) \rangle$ and $\langle (T1), (T4) \rangle$.

Observation about left memory nodes: If two tokens in a left memory node can be merged, that is, differ in only one slot, they differ in the last slot. Tokens arriving at a memory node are generated by appending a collection of tuples (from a right memory node) to a token (from a left memory node). Therefore, the collection in the last slot will always be different for all the tokens in the same left memory node. Now, if two tokens differ in at most one slot, they

must, necessarily, differ in the last slot. A corollary of this is that two mergeable tokens in a left memory node have a common token as a parent (the common subtoken consisting of all the slots except the last one).

Addition of a token: The memory node is searched for existing tokens that the incoming token can be merged with. If such a token is found, the two tokens are merged. Else, the new token is added to the memory node. For memory nodes associated with non-negated conditions, there can be at most one existing token in the memory node that can be merged with the incoming token. To show this, assume that there are two such tokens. Since they differ in only one slot *and* both of them belong to the same memory node, they differ in the last slot. Furthermore, both of them differ from the incoming token also in the last slot. Therefore, they can be merged with each other. Since tokens are added one at a time, one of them must have been added before the other at which time it would have been merged with it. Memory nodes associated with negated conditions need to maintain a count of matches in the opposite memory for every token. It is possible that two mergeable tokens match different tuples from right memory. The need to keep track of the matches in the opposite memory forces such tokens to be maintained as separate entities. For example, consider the production and tuple-space in Figure 6.3. The left memory corresponding to the only β node in the production contains the token $\langle \{T_1, T_2\} \rangle$. Since no tuples match the second condition, the opposite memory is empty and an instantiation is generated. Now suppose the tuple (request ^operation book-ticket ^priority 1) is added to the tuple-space. It matches one of the two tuples in $\langle \{T_1, T_2\} \rangle$. This forces a breach since the two tuples, T_1 and T_2 , match different tuples in the right memory.

```

(p make-request
  (operation-pending ^name <op>)
  -(request ^operation <op>)
  -->
  (make request ^operation <op> ^priority 1))

T1: (operation-pending ^name book-ticket)
T2: (operation-pending ^name buy-guidebook)

```

Figure 6.3: Example production with a negated condition.

Deletion of a token: Deletion of a token occurs either when a tuple matching a positive condition is deleted or a tuple matching a negative condition is created. While it is possible to mirror the addition operation for deletion, as is done in many production system implementations, it is

cheaper to scan the tokens for the tuple that initiated the deletion and remove it. For example, given a left memory node containing the tokens $\langle \{T1\}, \{T2, T3\} \rangle$ and $\langle \{T4\}, \{T3, T5\} \rangle$, deletion of the tuple T3 modifies the memory node to $\langle \{T1\}, \{T2\} \rangle, \langle \{T1\}, \{T5\} \rangle$. If the tuple being eliminated is the last one in any of the collections it occurs in, the corresponding token is deleted. For example, if after T3, T2 is deleted, the second slot in the first token becomes empty. The memory node now contains just $\langle \{T1\}, \{T5\} \rangle$.

Searching for a match: A left memory node is searched for a match for tuples from the corresponding right memory node. Conceptually, searching for a match consists of breaching every token in the memory node. Usually, only one of the two subtokens need be generated – the consistent subtoken if the β node performing the search corresponds to a non-negated condition and the inconsistent subtoken it corresponds to a negated condition.

6.2.3 Pnodes

Conceptually, a pnode contains the list of instantiations of a particular production. It is similar to a left memory node except that it uses the selection algorithm to impose an order on the instantiations. Ordering algorithms used for tuple-oriented instantiations can be easily extended to collection-oriented instantiations since all tuples in a collection are equivalent and the first tuple can be used as a proxy for all the tuples in the collection. The only complication arises if the selection algorithm uses a recency-based criterion that assigns unique timetags to tuples and favors higher values of the timetag over lower ones. Such selection schemes can be efficiently supported if the tuple with the highest timetag is the first one in the collection.

6.3 Collection Rete

This section describes Collection Rete, the collection-oriented analogue of Rete. The algorithm is presented in a pseudo-code form. Existence of the following mapping structures is assumed:

- *right_memories* maps a β nodeid to the contents of the corresponding right_memory node
- *left_memories* maps a β nodeid to the contents of corresponding left_memory node
- *pnodes* maps a pnode nodeid to its contents
- *successor_lists* maps a β nodeid to the list of nodeids of the successor nodes
- *node_type* maps a β or pnode nodeid to its type which may be AND, NOT or PNODE
- *beta_tests* maps a β nodeid to the corresponding inter-condition consistency test
- *right_memory_tests* maps a β nodeid to the test used to partition the corresponding right memory node into equivalence classes

```

procedure add_tuple(tuple)
begin
  right_memories ← find_matching_conditions(rete.net,tuple)
  /* find_matching_conditions() uses the upper half of the Rete network (the  $\alpha$  network) to
  find the set of conditions that match the given tuple. Every condition has an associated
  memory node which contains the tuples matching it */
  foreach memory in right_memories
    add_tuple_to_right_memory(tuple,memory)
  end
end

```

```

procedure add_tuple_to_right_memory(tuple,index)
begin
  classes ← right_memories[index]
  test ← right_memory_tests[index]
  /* See Section 6.2.1 for details on right_memory_tests */
  foreach class in classes
    result ← apply_test(test,tuple,class)
    /* apply_test() assumes that the values in the field can be ordered. It returns one of
    {BELONGS, LESS.THAN, GREATER.THAN} */
    if (result = BELONGS)
      add_tuple_to_class(tuple,class)
      return
    else if (result = GREATER.THAN)
      break
  end
  insert_class(create_class(tuple),right_memories[index])
  /* invoke the appropriate right activation routine */
  if (node.type[index] = AND)
    and_node_right.add(tuple,index)
  else if (node.type[index] = NOT)
    not_node_right.add(tuple,index)
  else
    add_tuple_pnode(tuple,index)
end

```

```

procedure and_node.right.add(tuple,index)
begin
  tokens ← left.memories[index]; test ← beta.tests[index]
  successors ← successor.lists[index]
  foreach token in tokens
    (token.con,token.incon) ← breach(token,tuple,test)
    if (token.con ≠ NULL)
      /* The consistent subtoken is extended and propagated down the network. In
      practice, only the consistent subtoken needs to be generated */
      succ_token ← extend(token.con,create.collection(tuple))
      foreach succ in successors
        add.to.left.memory(succ.token,succ)
    end
  end
end

procedure not_node.right.add(tuple,index)
begin
  tokens ← left.memories[index]; test ← beta.tests[index]
  foreach token in tokens
    (token.con,token.incon) ← breach(token,tuple,test)
    if (token.incon ≠ NULL)
      /* The original token is replaced by the breached subtokens, each with its own
      count of matches in the opposite memory. Breaching is required to maintain
      separate counts */
      delete_token(token,left.memories[index])
      add.token(token.incon,left.memories[index])
      add.token(token.con,left.memories[index])
      token.incon.matches ← token.matches
      token.con.matches ← token.matches+1
      /* If the inconsistent subtoken has no matches, neither did the original. Deleting
      all matching tuples from successor nodes is sufficient to replace successors of the
      original token by corresponding successors of token.incon */
      if (token.incon.matches = 0)
        delete.tuples.from.successors(matched.tuples(token.con,test),index)
      else
        token.matches ← token.matches+1
    end
  end
end

```

```

procedure and_node.left.add(token,index)
begin
  classes ← right.memories[index]
  test ← beta.tests[index]
  successors ← successor_lists[index]
  succ.tokens ← NULL

/* This routine scans the associated right memory and generates a sequence of successor tokens. In effect, it breaches the original token for every equivalence class and then merges as many of them as possible. In practice, it is possible to avoid the merging step by taking advantage of the fact that in an overwhelming fraction of cases, there is only one test, usually an equality test */

foreach class in classes
  (token.con,token.incon) ← breach(token,class.first_tuple,test)
  new_token ← extend(token.con,class)
  merged ← FALSE
  foreach succ_token in succ.tokens
    if (mergeable(new_token,succ_token))
      merge(new_token,succ_token)
      merged ← TRUE
  end
  /* If the new token cannot be merged, add it to the list of successors */
  if (merged = FALSE)
    add_token(new_token,succ.tokens)
end

/* Propagate all the generated tokens to all the successor nodes */
foreach succ in successors
  foreach succ_token in succ.tokens
    add_to_left_memory(succ_token,succ)
  end
end
end

```

```

procedure not_node_left_add(token,index)
begin
  classes ← right_memories[index]
  test ← beta_tests[index]
  successors ← successor_lists[index]
  consistent_tokens ← NULL
  inconsistent_tokens ← NULL

/* This routine scans the associated right memory and generates a sequence of successor tokens. The sense of tests is reversed from and_node_left_add(). It keeps track of the subtokens generated and the number of their matches. The original token is replaced by the subtokens in the left memory for this node */

  foreach class in classes
    (token.con,token.incon) ← breach(token,class.first_tuple,test)

    if (find_token(token.con,consistent_tokens,1) = NOT_FOUND)
      token.con.matches ← 1
      add_token(token.con,consistent_tokens)
    if (find_token(token.incon,inconsistent_tokens,0) = NOT_FOUND)
      token.incon.matches ← 0
      add_token(token.incon,inconsistent_tokens)
  end

/* Propagate all the generated tokens to all the successor nodes */
  foreach succ in successors
    foreach incon_token in inconsistent_tokens
      add_to_left_memory(extend(incon_token,NULL),succ)
    end
  end

/* Replace the original token by the lists of subtokens */
  delete_token(token,left_memories[index])
  append_token_list(consistent_tokens,left_memories[index])
  append_token_list(inconsistent_tokens,left_memories[index])
end

```

```

procedure add_to_left_memory(incoming_token,index)
begin
  tokens ← left_memories[index]
  merged ← FALSE
  foreach token in tokens
    if (mergeable(incoming_token,token))
      /* mergeable() can be efficiently implemented if a canonical order can be imposed
         on the tuples in each collection. The test for equality for collections can be
         implemented as p/r equality on the first tuple */
      merge(incoming_token,token)
      merged ← TRUE

    /* For AND nodes, tokens have maximal collections, since there is no need to
       maintain counters for subcollections. Therefore, there is only one mergeable
       token. */
    if (node.type[index] = AND)
      break
  end
  /* If there is no mergeable token, add the token to the memory node */
  if (merged = FALSE)
    add_token(token,left_memories[index])
  /* Left activation of the successor node */
  if (node.type[index] = AND)
    and_node_left.add(token,index)
  else if (node.type[index] = NOT)
    not_node_left.add(token,index)
  else
    add_token_to_pnode(token,index)
  end

procedure find_token(incoming_token,token_list,incr)
begin
  /* If a match is found, the existing token's match count is incremented by incr */
  foreach token in token_list
    if (identical_token(incoming_token,token))
      token.matches ← token.matches+incr
      return (FOUND)
  end
  return (NOT_FOUND)
end

```

```

procedure delete_tuples_from_successors(tuples,index)
begin
  successors ← successors_list[index]

  foreach succ in successors
    foreach tuple in tuples
      if (node_type[index] = AND or node_type[index] = NOT)
        delete_tuple_left_memory(tuple,succ)
      else
        delete_tuple_pnode(tuple,succ)
    end
  end
end

procedure add_tuple_pnode(tuple,index)
begin
  /* This routine is invoked when the production has only one condition. Therefore,
  all tuples that reach the right memory node corresponding to the condition, match the
  production. In this case, all instantiations can be merged into a single collection-oriented
  instantiation. This routine creates a unit collection for the given tuple and merges it into
  the single collection-oriented instantiation */
  merge_inst(creat_inst(create_collection(tuple)),pnodes[index])
end

procedure add_token_to_pnode(token,index)
begin
  /* This routine is invoked for productions with multiple conditions and therefore multiple
  instantiations are possible. Similar to an and  $\beta$  node, the instantiations are maximal
  and every incoming token can be merged with only one of them. */
  instantiations ← pnodes[index]
  foreach inst in instantiations
    if (mergeable_inst(token,inst))
      merge_inst(token,inst)
    return
  end
  /* The token can be merged with none of the existing instantiations. Create a new
  instantiation and add it to the conflict set. Note that instantiations that have already
  been fired are held separately and are not a part of the conflict set */
  add_inst(create_inst(token),pnodes[index])
end

```

```

procedure delete_tuple(tuple)
begin
    right_memories ← find_matching_conditions(rete_net,tuple)
    foreach memory in right_memories
        delete_tuple_right_memory(tuple,memory)
    end
end

procedure delete_tuple.right_memory(tuple,index)
begin
    classes ← right_memories[index]
    /* assumption: that a tuple can occur in only one equivalence class */
    foreach class in classes
        if (apply_test(test,tuple,class) = BELONGS)
            delete_tuple_from_class(tuple,class)
            if (empty(class))
                delete_class(class,right_memories[index])
                break
        end
    /* Propagate the deletion */
    if (node.type[index] = AND)
        and_node_right_delete(tuple,index)
    else if (node.type[index] = NOT)
        not_node_right_delete(tuple,index)
    else
        delete_tuple_pnode(tuple,index)
    end

procedure and_node_right_delete(tuple,index)
begin
    successors ← successor_lists[index]
    /* Iterator to delete the tuple from all successors */
    foreach succ in successors
        if (node.type[index] = AND or node.type[index] = NOT)
            delete_tuple_left_memory(tuple,succ)
        else
            delete_tuple_pnode(tuple,succ)
        end
    end
end

```

```

procedure not_node.right.delete(tuple,index)
begin
  tokens ← left.memories[index]; test ← beta.tests[index]
  successors ← successor_lists[index]
  foreach token in tokens
    /* match_token() checks if the entire token is consistent with tuple */
    if (match_token(token,tuple,test))
      token.counter ← token.counter-1
      if (token.counter = 0)
        /* No matches left, need to generate successor token */
        succ_token ← extend(token,NULL)
        foreach succ in successors
          if (node.type[succ] = AND or node.type[succ] = NOT)
            delete_tuple_left_memory(tuple,succ)
          else
            delete_tuple_pnode(tuple,succ)
        end
    end
  end

procedure delete_tuple_left_memory(tuple,index)
begin
  tokens ← left.memories[index]
  successors ← successor_lists[index]
  foreach token in tokens
    foreach slot in slots(token)
      delete_tuple_from_slot(tuple,slot)
    /* If any of the slots becomes empty, then there is no match left */
    if (empty(slot))
      delete_token(token,left.memories[index])
      break
    end
  end
  foreach succ in successors
    if (node.type[index] = AND or node.type = NOT)
      delete_tuple_left_memory(tuple,succ)
    else
      delete_tuple_pnode(tuple,succ)
  end
end

```

```
procedure delete_tuple_pnode(tuple,index)
begin
  instantiations ← pnodes[index]
  foreach inst in instantiations
    foreach slot in slots(inst)
      delete_tuple_from_slot(tuple,slot)
      if (empty(slot))
        delete_instantiation(inst,pnodes[index])
        break
    end
  end
end
```

Chapter 7

Collection-oriented Production Language

Tuple-oriented production languages operate on scalars - each condition matches at most one tuple, each variable is bound to a single value and each tuple-space operation creates or modifies a single tuple. The semantics of these languages require that the instantiations generated at the end of the match should be tuple-oriented. This implies that while it is possible to use collection-oriented match algorithms for implementing these languages, the collection-oriented instantiations generated must be converted to their tuple-oriented analogues before they can be used. In many cases, however, there is no need to generate the cross-products. Therefore, these languages cannot take full advantage of collection-oriented match algorithms.

Another limitation of tuple-oriented languages is their inability to express aggregate operations on collections of data. Tuple-oriented languages that fire multiple instantiations per *msa* cycle, like PPL and PARULEL, allow only element-wise operations. In addition to limiting expressiveness, this restriction creates serious performance problems for tight sequential loops. In such cases, it is possible that the matching overhead can swamp the computation.

This chapter presents the design and implementation of a collection-oriented production language, COPL.

7.1 Design of the Collection-oriented Production Language

7.1.1 Desiderata

From the discussion in the previous chapter, it is clear that the primary goal in the design of a collection-oriented production language should be to allow the programmer (and the compiler) to delay and, if possible, eliminate cross-products between collections of tuples matching different conditions. To delay cross-products, it is necessary to create and hold the component

collections. This implies that such a language should allow conditions to be matched by collections of tuples instead of individual tuples and correspondingly it should generalize the notion of instantiation from a sequence of tuples to a sequence of collections of tuples.

Recall from Section 2.2, that variables are bound on their first occurrence. The condition in which a variable first occurs is referred to as its *binding condition* and the field in which it appears is referred to as its *binding field*. In scalar languages, like OPS5, a variable is bound to the value of the binding field in the tuple that matches the binding condition. For example, in Figure 7.1, the variable `<var>` is bound to the value from the `size` field of the tuple(s) that match(es) the first condition. In a scalar language, the tuple-space in Figure 7.1 results in the generation of three instantiations, each with a separate tuple corresponding to the first condition and accordingly, `<var>` is separately bound to 11, 12 and 13. In a collection-oriented language, a variable is bound to the value of the binding field for *all* the tuples that match the binding condition. For example, in Figure 7.1, the binding condition for the variable `<var>`, that is the first condition in the production, matches the collection `<{T3, T4, T5}, {T1}>`. As a result, `<var>` is bound to the collection of `size` values from all the tuples that match this condition, that is {11,12,13}.

```
(p jack-boots
  (object ^type jack-boots ^size <var>)
  (specification ^object jack-boots ^size <= <var>)
  -->
  (modify 1 ^acceptable yes))

T1. (specification ^object jack-boots ^size 11)
T2: (object ^type jack-boots ^size 10)
T3: (object ^type jack-boots ^size 11)
T4: (object ^type jack-boots ^size 12)
T5: (object ^type jack-boots ^size 13)
```

Figure 7.1: Example to illustrate variable binding

To take the full advantage of collection-valued variables, a collection-oriented production language must extend the actions in the then-part to operate on collections. For example, in Figure 7.1, the `modify` action should mark *all* the tuples matching the first condition as acceptable.

The following subsections describe the design of the Collection-oriented Production Language (COPL). This language has been based on OPS5. Details on OPS5 can be found in Section 2.2.

7.1.2 Tuple space

Given the emphasis on collections, allowing tuples to contain collection-valued fields is a natural extension. And if fields are allowed to contain collections, there is no reason why they should be restricted to flat collections. The main argument against allowing collection-valued fields is that they would introduce structure in the tuple-space and, in the extreme case, cause the tuple-space to collapse into a single monolithic tuple. Such a drastic change in the data representation would make it difficult, if not impossible, to compare the performance of collection-oriented languages and the corresponding match algorithms with their tuple-oriented analogues. Furthermore, such a language would be incompatible with at least one of the motivating applications – active relational databases. As a result, COPL retains an unstructured tuple-space.

7.1.3 Conditions and Instantiations

COPL makes no changes in the OPSS syntax for conditions. Each condition is matched by a collection of tuples. Instantiations consist of a list of collections, one collection corresponding to each condition in the production. The collections corresponding to non-negated conditions must have at least one tuple; the collections corresponding to negated conditions must be empty. Instantiations are *maximal*, that is it is not possible to add a tuple to any of its component collections without violating the mutual consistency requirement.

Each variable is bound to the collection of values from the binding field. Values matching all occurrences of a variable must be mutually consistent. Figure 7.2 shows an example. This production attempts to pair up requests for operations with servers that are capable of performing the operation. The first instantiation corresponds to requests and servers for the *current-time* operation and the second instantiation corresponds to the *set-time* operation. Note that the instantiations are maximal.

Since individual COPL instantiations usually correspond to a large number of OPSS instantiations, the COPL conflict sets are much smaller. Therefore, the selection strategy used is much less important. To order the collection-oriented instantiations, COPL extends the OPSS selection strategies. Recency of an instantiation is computed by extracting the most recent tuple in every collection and using it as a proxy for the entire collection. Figure 7.3 shows how the instantiations in Figure 7.2 would be ordered.

7.1.3.1 Actions

Extending actions to deal with collection-valued variables is more involved. Consider, for example, the production in Figure 7.4. Under tuple-oriented semantics, a separate instantiation

```

(p handle-requests
  (request ^id <id> ^operation <op>)
  (server ^name <server> ^capable-of <op>)
  -->
  (some actions))

T1. (request ^id 1 ^operation current-time)
T2. (request ^id 2 ^operation current-time)
T3. (request ^id 3 ^operation set-time)
T4: (server ^name time-0 ^capable-of set-time)
T5: (server ^name time-1 ^capable-of current-time)
T6: (server ^name time-2 ^capable-of current-time)

Instantiations:
<(T1,T2),(T5,T6)> -- <op> = current-time, <id> = {1,2}, <server>= {time-1,time2}
<(T3),(T4)> -- <op> = set-time, <id> = {3}, <server> = {time-0}

```

Figure 7.2: Example of binding collections to variables

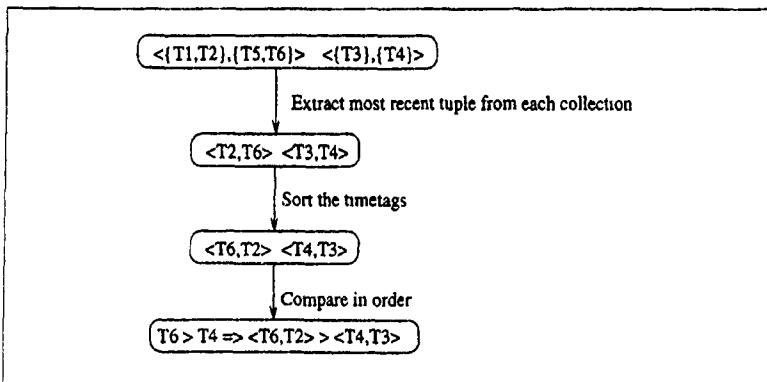


Figure 7.3: Example of instantiation ordering in COPL

is generated for every way in which the production can be matched (there are nine such instantiations). The make action for each instantiation has unique values for both $\langle obj1 \rangle$ and $\langle obj2 \rangle$. Under collection-oriented semantics, only one instantiation is generated and both the variables are bound to the collection {T1,T2,T3}. To achieve the same functionality as the tuple-oriented version of the program, the make action has to generate a cross-product of the two variables.

```
(p make-pairs
  (object ^id <obj1> ^type <t1>)
  (object ^id <obj2> ^type <t2>)
  ->
  (make pair ^first <obj1> ^second <obj2>))

T1: (object ^id 1 ^type abacus)
T2: (object ^id 2 ^type abacus)
T3: (object ^id 3 ^type abacus)
```

Figure 7.4: Production that generates a cross-product in the tuple-space

However, generating a cross-product of variables is not appropriate for all instances of a make action. Consider the case in Figure 7.5. Under tuple-oriented semantics, again nine instantiations are generated. But only three of them are fired as each firing also deletes one of the objects. The result of the three firings is three pair tuples, each with a different object in the first field. To achieve the same functionality under collection-oriented semantics, the make action should establish a 1-to-1 correspondence between the values of $\langle obj1 \rangle$ and $\langle obj2 \rangle$.

To correctly handle the conflicting requirements of these two situations, COPL extends the semantics of make and introduces two new functions - insert and update. The make action creates a cross-product of all argument collections but it does not add them to the tuple-space. Instead, it gathers them into a collection and returns the collection as the result of the make action. Actual addition of the tuples to the tuple-space is done by insert. The update function takes three arguments, a collection of tuples, a field index and a collection of values. The cardinality of the two collections should be identical. It assigns the n^{th} value in the second collection to the given field of the n^{th} tuple in the first collection. It returns the collection of tuples after the update. Figure 7.6 shows how these functions can be used to write COPL versions of the productions shown in Figures 7.4 and 7.5.

The treatment of the modify action is simpler. This action only updates fields of tuples and

```
(p make-pairs-unique
  (object ^id <obj1> ^type <>)
  (object ^id <obj2> ^type <>)
  -->
  (make pair ^first <obj1> ^second <obj2>)
  (remove 1))

T1: (object ^id 1 ^type abacus)
T2: (object ^id 2 ^type abacus)
T3: (object ^id 3 ^type abacus)
```

Figure 7.5: Production that does not generate a cross-product in the tuple-space

```
(p make-pairs
  (object ^id <obj1> ^type <>)
  (object ^id <obj2> ^type <>)
  -->
  (insert (make pair ^first <obj1> ^second <obj2>)))

(p make-pairs-unique
  (object ^id <obj1> ^type <>)
  (object ^id <obj2> ^type <>)
  -->
  (insert (update (make pair ^first <obj1>
    ^second <obj2>))
  (remove 1)))
```

Figure 7.6: Cross-product and non-cross-product makes in COPL

never generates additional tuples. COPL allows the arguments to `modify` be either scalar values or collections. However, the cardinality of any collection-valued argument must be the same as the cardinality of the collection of tuples being modified. Figure 7.7 shows an example of the `modify` action. This production computes the sum of all `values` and stores in the `sum` tuple. Note that the call to the function `sum` takes a collection as an argument.

```
(p sum-values
  (value ^data <d>)
  (sum ^result 0)
  -->
  (modify 1 ^result (sum <d>)))
```

Figure 7.7: Example of `modify` in COPL

The `remove` action can be easily extended to handle collections. Instead of deleting a single tuple, it deletes a collection of tuples.

The foreign functions called by COPL can take either scalar or collection-valued arguments.

The next section describes `coplc`, an implementation of COPL.

7.2 Implementation of COPL

`coplc` is an implementation of a subset of COPL. It compiles all of COPL except negated conditions to portable C code. The run-time system is entirely in portable C. It requires no special operating system support and runs on any machine that has a C compiler. `coplc` generates uniprocessor code and makes no attempt to parallelize the program. For matching, it uses the Collection Rete algorithm described in Section 6.3. `coplc`, is based on the PPL implementation described in Section 3.4. The `pplc` compiler has been modified to generate code for the Collection Rete algorithm. The run-time library has been modified to handle collection-oriented tokens and instantiations and to support the extended versions of the actions. The description of the Collection Rete algorithm in Section 6.3 leaves several operations, like `mergeable()`, `breach()` and `create_inst()`, unspecified in the interest of clarity. Following subsections describe how these operations have been implemented.

7.2.1 Data structures

Right memory: A right memory is implemented as an ordered list of equivalence classes. Each equivalence class is implemented as an ordered list of tuples. The first field being tested at the β node associated with the memory is used to partition the tuples into equivalence classes. This field is referred to as the *characteristic* field of the memory. An equivalence class contains all tuples with the same value for the characteristic field. Since most β nodes have a single test, and since membership tests are expected to be frequent, equivalence classes cache the value of the characteristic field. The order on the values in the characteristic field is used to order the equivalence classes. The tuples within each equivalence class are ordered in the descending order of their timetags.

Ordering the equivalence classes allows the search for an equivalence class to often be terminated without traversing the entire list. This allows efficient implementations of addition and deletion. It also allows efficient implementation of searching for matches. Searching for an equality test consists of finding the equivalence class with a matching characteristic value; searching for an inequality test consists of collecting the tuples in all the other equivalence classes; searching for relational tests consists of collecting the tuples in all the equivalence classes before (for $<$ tests) or after (for $>$ tests) the matching equivalence class.

`cop1c` does not hash right memories since partitioning the tuples into equivalence classes already achieves most of the benefits of hashing. If the number of equivalence classes becomes large, a hybrid scheme which uses hashing to quickly find desired equivalence classes might become attractive.

Left memory: A left memory is implemented as a list of collection-oriented tokens. Each token is an array of collections, one collection corresponding to each condition. Each collection is an ordered list of pointers to tuples. Pointers to tuples are ordered by the timetags of the tuples they point to. Ordering the tuples permits efficient equality tests on collections. Two collections are equal if and only if the first tuple in both of them is the same. Therefore, the test for equality of collections reduces to the test for the equality of a pair of pointers. Efficient implementation of collection equality, in turn, allows efficient implementations of procedures that manipulate tokens like, `mergeable()` and `find_token()`.

Hashing left memories has been shown to achieve a significant speedup for Rete [42] since it quickly isolates the tokens that are likely to match. There are two reasons why hashing is unlikely to achieve similar gains for Collection Rete. First, since each collection-oriented token usually corresponds to a large number of tuple-oriented tokens, the size of left memories is expected to be much smaller for Collection Rete. Second, since the tuples in a collection can have different values for the field(s) being tested at the associated β node, a collection-oriented token would have to be hashed to multiple buckets. Partitioning a collection-oriented token so that each partition only has tuples with the same value(s) for field(s) to be tested would allow each token to be hashed to a single bucket but would violate the maximality requirement for

instantiations. Reordering the tuples based on the value(s) of the field(s) to be tested would destroy the canonical order based on timetags. Without a canonical order on the tuples in a collection, testing two collections for equality would require complete traversals of both collections.

Conflict set: The conflict set consists of a heap of heaps. All the instantiations for individual production are organized in a production-specific heaps; the highest ranking instantiations for all the production are organized into another heap. As mentioned earlier, COPL uses an extended version of the OPS5 instantiation selection strategy. Recency of an instantiation is computed by extracting the most recent tuple in every collection and using it as a proxy for the entire collection. Figure 7.3 shows how the instantiations in Figure 7.2 would be ordered.

7.2.2 Procedures

mergeable (token1, token2) : Two tokens can be merged if and only if they differ in at most one collection. As discussed above, equality of two collections can be tested by checking if the first pointer in the two lists are equal. Therefore, checking if two tokens can be merged needs at most n pointer comparisons where n is the number of collections.

merge (token1, token2) : This procedure merges the first token into the second. It is assumed that the two tokens differ in only one slot and the index of this slot has already been computed (presumably during the call to `mergeable()` which precedes the call to this procedure). The first token is merged into the second by merging its collection in the differing slot with corresponding collection for the second token. This requires merging of two sorted lists of tuples. The cost of merging two sorted lists of length, l_1 and l_2 is $O(l_1 + l_2)$.

find_token (token, token.list, incr) : This procedure searches `token.list` for `token`. Two tokens are equal if and only if all their collections are equal. Therefore, finding a token takes $O(nl)$ time where n is the number of collections in the tokens and l is the length of the list of tokens. If the token is found, its count of matches is incremented by `incr`.

extend (token, tuple) : This procedure is used to create successor tokens. The existing token is copied and a singleton collection consisting of `tuple` is appended to it. Since only the pointers to the collections have to be copied and not entire collections, the time taken is $O(n)$ where n is the number of collections. Creation of the singleton collection and appending it to the token takes constant time since a token is implemented as an array of collections.

delete_tuple_from_slot (tuple, slot) : This procedure first checks if the tuple to be deleted and the tuples in the slot are of the same type. If not, it returns right away. Otherwise, it takes advantage of the timetag order on the tuples in a collection to avoid scanning tuples whose timetags are smaller than its own. In the worst case, it may need to traverse the whole collection.

`breach(token, tuple, test)` : This procedure splits `token` into two parts, one that is consistent with `tuple` and the other that is inconsistent with it. Consistency with a tuple is defined in terms of `test` which tests one or more fields of the tuple and one or more slots of the token. The consistent and inconsistent subtokens differ only in the slots that are tested by `test`. Breaching is implemented by scanning the slots tested by the `test` and partitioning their collections into consistent and inconsistent subcollections. The subtokens are created by copying the pointers to slots that are not tested and copying the subcollections for the slots that are. Copying a collection consists of copying the list of pointers to tuples. The tuples themselves are not copied. If k slots out of n are tested and the lengths of the subcollections in the i^{th} tested slot l_i , then the time taken is $(n - k) + \sum l_i$.

`create_inst(token)` : this procedure extracts the most recent timetag from each of the collections in the instantiation and sorts them. Since, the collections are sorted in descending order of timetags, extracting the most recent timetag for a collection is a constant time operation. It uses heap sort to sort the timetags which takes time $O(n \log n)$ where n is the length of the instantiation.

The next chapter evaluates the scalability of collection-oriented match algorithms and compares their performance with that of tuple-oriented algorithms for large tuple-spaces.

Chapter 8

Collection-oriented Match Experiments

The primary goal of these experiments was to evaluate the scalability of collection-oriented match algorithms. A lesser goal was to compare the performance of collection-oriented match algorithms and their tuple-oriented analogues for large tuple-spaces. To help achieve both goals, programs that process scalable data sets were selected as benchmarks and each benchmark program was run with successively larger tuple-spaces. In these experiments, Rete was selected as the exemplar for tuple-oriented match algorithms and its collection-oriented analogue, Collection Rete, was selected as the exemplar for collection-oriented match algorithms. The corresponding compilers, `pplc` and `copl`, differ only in the match algorithm used. The run-time libraries for the two implementations differ only in COPL's support for aggregate operations on collections. In comparison, PPL provides only element-wise operations on collections through joint firing of all instantiations of a production. Therefore, the differences in performance can be directly attributed to these two factors. This chapter describes the experiments and their results. The first section describes the benchmark programs. The next section describes the structure of the experiments. The third section compares the performance of PPL and COPL on the benchmark programs. The fourth section discusses the scalability of collection-oriented match algorithms. The chapter concludes with some observations from the experiments including programming idioms for collection-oriented production languages.

8.1 Benchmarks

The benchmark suite consists of three programs, `make-teams`, `clusters` and `airline-route`. Two of them, `make-teams` and `airline-route` operate on databases (of employees and airline routes respectively) and the third, `clusters`, performs clustering on image regions. All of them process scalable data sets, that is data sets that can be characterized by a numerical

parameter and which can be scaled by assigning increasing values to this parameter. All of them were originally written in OPS5. Code for these programs can be found in Appendix F.

8.1.1 Creating teams with constraints (make-teams)

This program operates on a database of employees which contains information about their area of expertise and previous experience. It also contains an overall numerical evaluation of each employee's past performance. The task is to build teams with four employees each. The teams are to be formed with the constraints that each member must have a different area of expertise and that some of the members must have worked together previously. There are four areas of expertise – hardware, compilers, networks and operating systems. This program builds all such teams and determines the number of teams that are "good". The "goodness" of a team is defined as a sum of the evaluations of its members. This program was written by Milind Tambe at Carnegie Mellon University.

This program consists of three phases. The first phase creates all valid teams and computes their "goodness" score. Each team can be independently created. The second phase identifies all the "good" teams based on the "goodness" score computed by the previous phase. The final phase counts the number of "good" teams.

Conversion to PPL: Since all teams can be created independently, the production that creates the teams and computes their goodness score was converted to a parallel production. Since selection of a team as a "good" team depends only on its own score, it can be performed in parallel for all the teams. To achieve this, the production that selects "good" teams was converted to a parallel production. The third phase consists of a non-parallelizable loop similar to the counting loops in hotel (see Section 4.1.4). PPL versions of these three productions are shown in Figure 8.1.

Conversion to COPL: Only three productions in this program have conditions that are matched by more than one tuple – the production that creates the teams, the production that selects "good" teams and the production that counts the selected teams. Consider the first production (its PPL version is shown in Figure 8.1). Under tuple-oriented semantics, there are as many instantiations as teams. Each instantiation creates the tuple corresponding to one team and computes its score. Under collection-oriented semantics, there are as many instantiations as the number of previous projects. Each instantiation creates the tuples for all the teams whose hardware and compiler experts have previously worked together on a particular project (the value bound to the variable `<project>`). The tuples for all such teams can be created by generating the cross-product of the collections bound to `<id1>`, `<id2>`, `<id3>` and `<id4>` and creating a tuple for every element of the cross-product. This is exactly what the COPL make action does. This production also computes the "goodness" score for each team. To achieve that, the COPL version of this program uses a C procedure `compute4()` which generates a

```

(parl make-team
  (goal ^name create-teams)
  (person ^id <id1> ^expertise hardware ^previous-project <project> ^score <v1>)
  (person ^id <id2> ^expertise operating-system ^score <v2>)
  (person ^id <id3> ^expertise networks ^score <v3>)
  (person ^id <id4> ^expertise compilers ^previous-project <project> ^score <v4>)
  -->
  (make team ^hardware <id1> ^operating-systems <id2> ^networks <id3>
        ^compilers <id4> ^score (compute <v1> + <v2> + <v3> + <v4>))

(parl create-teams
  (goal ^name select-team)
  (team ^score > 8 ^select-status nil)
  -->
  (modify 2 ^select-status selected))

(parl count-teams
  (goal ^name count-teams)
  (team ^select-status selected)
  (count ^value <value>)
  -->
  (modify 2 ^select-status counted)
  (modify 3 ^value (compute <value> + 1)))

```

Figure 8.1: PPL productions for make-teams

corresponding cross-product of the individual scores (collections bound to <v1>, <v2>, <v3> and <v4>) and sums each combination.

The production that selects “good” teams needs no conversion as all it does is match a set of tuples and modify them. Under tuple-oriented semantics, a separate instantiation is generated for every team whereas under collection-oriented semantics only one instantiation is generated for all the teams.

As discussed in Section 5.2.4, loops that perform any sort of accumulation over a collection of data items are a major source of inefficiency in tuple-oriented production system languages. The

production that counts the selected teams is an instance of this. In every cycle, an instantiation is created for every team that has not yet been counted. One of these instantiations is selected and fired. This modifies the tuple containing the counter which leads to the deletion of the other instantiations. In the next *msa* cycle, this process repeats with the new counter tuple and the remaining teams. As a result, $O(n^2)$ instantiations are created, of which only n instantiations are fired. Under collection-oriented semantics, only one instantiation is generated and the teams are counted by a C function (*cardinality()*).

COPL versions of these three productions are shown in Figure 8.2.

```
(p make-team
  (goal ^name create-teams)
  (person ^id <id1> ^expertise hardware ^previous-project <project> ^score <v1>)
  (person ^id <id2> ^expertise operating-system ^score <v2>)
  (person ^id <id3> ^expertise networks ^score <v3>)
  (person ^id <id4> ^expertise compilers ^previous-project <project> ^score <v4>)
-->
  (insert (update (make team ^hardware <id1> ^operating-systems <id2>
                           ^networks <id3> ^compilers <id4>)
                  ^score (compute4 <v1> <v2> <v3> <v4>)))))

(parp create-teams
  (goal ^name select-team)
  (team ^score > 8 ^select-status nil)
-->
  (modify 2 ^select-status selected))

(p count-teams
  (goal ^name count-teams)
  (team ^hardware <id> ^select-status selected)
-->
  (insert (make count ^value (cardinality <id>))))
```

Figure 8.2: COPL productions for make-teams

Data set: The data set for make-teams is parameterized by the number of employees. The

number of previous projects is fixed at ten and the number of compiler experts is fixed at five. The compiler and hardware experts are equally (and randomly) distributed over all the projects. Other employees are randomly assigned projects with no restriction on distribution. The restriction on the number of compiler experts was introduced to limit the number of tuple-oriented instantiations generated. Without this restriction, the number of tuple-oriented instantiations grew so fast that only very small data sets could be processed on a Decstation 5000/200 with 96 megabytes of main memory.

8.1.2 Clustering image regions (clusters)

This program operates on image regions that are characterized by position and type (e.g. road, hangar, tarmac, etc.). The regions are divided into *seed* regions, tarmacs, parking-aprons or hangars, and non-*seed* regions. Every *seed* region forms the center of a cluster which includes all regions within a given distance from it. The task is to determine the average size of these clusters. The computation performed is loosely similar to the computation in the second and third phases of SPAM[75],¹ a knowledge-based image analysis program. This program was written by Milind Tambe at Carnegie Mellon University.

This program consists of three phases. The first phase computes the distance between each *seed* region and all other regions. The second phase creates clusters around each *seed* region. The third phase computes the sizes of all clusters and computes their mean.

Conversion to PPL: The distance between all pairs of regions can be computed in parallel. To achieve this, the production that computes the distance between a pair of regions was converted to a parallel production. Since there is no restriction on overlapping of clusters, all clusters can be created in parallel. To achieve this, the production that generates the links between the *seed* and the other members of a cluster was converted to a parallel production. The third phase consists of two accumulation loops. The first loop is doubly nested and counts the number of members in every cluster. The second loop adds up these counts. Since PPL does not support aggregate operations like accumulation, productions in the third phase are left unchanged. Figure 8.3 shows the PPL versions of the converted productions.

Conversion to COPL: There are four productions in this program with conditions that match multiple tuples – the production that computes the distances, the production that creates the clusters, the production that computes the size of individual clusters, and the production that sums the sizes of the clusters. Consider the first production (its PPL version is shown in Figure 8.3). Under tuple-oriented semantics, a separate instantiation is generated for every pair of regions, the first being a *seed* region. Each instantiation computes the distance between this pair of regions and generates the corresponding *distance* tuple. Under collection-oriented

¹Section 4.1.5 contains a brief description of SPAM.

```

(parl compute-distance
  (goal ^name calculate-distance)
  (object ^number <n1> ^x <x> ^y <y> ^type <<tarmac parking-apron hangar-building>>)
  (object ^number <n2> ^x <x1> ^y <y1>)
  -->
  (make distance ^seed <n1> ^element <n2>
    ^value (compute (<x1>-<x>)^(<x1>-<x>) + (<y1>-<y>)^(<y1>-<y>)))

(parl create-clusters
  (group ^name create-groups)
  (object ^number <n1> ^focus yes)
  (distance ^seed <n1> ^element <n2> ^value (>0 <800))
  -->
  (make group ^center <n1> ^member <n2> ^counted no))

```

Figure 8.3: PPL productions for clusters

semantics, only one instantiation is generated, *<n1>* being bound to the collection of all seed regions and *<n2>* being bound to the collection of all regions. To generate all the *distance* tuples, it is necessary to create a cross-product of *<n1>* and *<n2>*. This can be directly done using the COPL *make* action. The distance between different pairs of points is computed by the C function *compute_distances()* which creates a corresponding cross-product of the coordinates and returns a collection of distances (actually squares of distances).

The production that creates the clusters is simpler to convert. Under collection-oriented semantics, one instantiation is generated per seed region, *<n1>* is bound to the seed region and *<n2>* is bound to the collection of regions within $\sqrt{800}$ distance units from it. One link is to be created between a seed region and every member of its cluster. Creating a cross-product of *<n1>* and *<n2>* using the *make* action achieves the desired effect.

The remaining two productions accumulate values from collections. Their collection-oriented versions move the accumulation loops from the production system paradigm where they are inefficient to a procedural paradigm where they can be efficiently implemented. The COPL versions of these productions use two C procedures, *summation()* and *cardinality()*, for the computation.

Figure 8.4 shows the converted productions.

Data set: The data set for *clusters* is parameterized by the number of seed regions. The

```
(p compute-distance
  (goal ^name calculate-distance)
  (object ^number <n1> ^x <x> ^y <y> ^type <<tarmac parking-apron hangar-building>>)
  (object ^number <n2> ^x <x1> ^y <y1>)
  -->
  (insert (update (make distance ^seed <n1> ^element <n2>)
    ^value (compute_distances <x1> <x> <y1> <y>)))))

(p create-clusters
  (goal ^name make-groups)
  (object ^number <n1> ^focus yes)
  (distance ^seed <n1> ^element <n2> ^value (>0 <800)))
  -->
  (insert (make group ^center <n1> ^member <n2>)))

(p compute-cluster-size
  (goal ^name get-group-sizes)
  (object ^number <n1> ^focus yes)
  (group ^center <n1> ^member <n2>)
  -->
  (insert (make group-count ^center <n1> ^size (cardinality <n2>)))))

(p count-clusters-and-sum
  (goal ^name average-group-sizes)
  (group-count ^center <n1> ^size <sz>)
  -->
  (insert (make average-size ^sum (summation <sz>) ^count (cardinality <n1>))))
```

Figure 8.4: COPL productions for clusters

total number of regions in a data set is ten times the number of seed regions. Each region is randomly assigned a unique position in a 100x100 grid.² The relation between the number of seed and non-seed regions has been picked to keep the number of tuple-oriented instantiations under control.

8.1.3 Airline routing (airline-route)

This program operates on an airline flight database with fields for source, destination and cost of each flight. The task is to find a minimum cost route from a given source to a given destination with a given number of hops. If no route with the given number of stops can be found, the cheapest route overall is desired. This program was written by Milind Tambe at Carnegie Mellon University.

This program has two phases. The first phase computes all the routes from the source to the destination and the second phase picks the best available route.

Conversion to PPL: Finding a route in the database does not need to modify the database. Therefore, it is possible to compute all routes independently. This can be achieved by converting all productions that compute routes to parallel productions. Figure 8.5 shows the PPL version of one of them, the production that finds all the routes from the source to the destination with exactly one intermediate stop. The selection of the best available route is done using the match process directly. The production implementing this is also shown in Figure 8.5.

Conversion to COPL: The only productions whose conditions match multiple tuples are the productions that compute the routes. As an exemplar, consider the production that computes the routes with two hops (PPL version of this production is shown in Figure 8.5). Under tuple-oriented semantics, a separate instantiation is generated for every pair of flights that connect the source and destination. Under collection-oriented semantics, one instantiation is generated per stop-over point; `<id1>` is bound to the collection of flights from the source to the stop-over point and `<id2>` is bound to the collection of flights from the stop-over point to the destination. All the corresponding route tuples can be generated by creating a cross-product of these two variables. The COPL version of the production (shown in Figure 8.6) uses a C function, `compute_costs_2()`, to compute the costs of all the flight combinations. It uses another C function, `vector_gensym()` to generate a vector of unique identifiers for the routes.

Data set: The data set for `airline-route` is parameterized by the number of flights from each hub airport. The number of airlines is fixed at ten and the number of airports is fixed at twenty. Each airline is randomly assigned an airport as a hub. The flight database is created by generating paired flights from these hubs to random destinations. The cost for each flight is randomly assigned and is same in both directions.

²A few of the very large data sets were generated in a 400x400 grid

```

(parl two-hops
  (goal ^name compute-routes)
  (traveller ^name <x> ^source <src> ^destination <dest>)
  (flight ^source <src> ^destination <stop-over> ^cost <c1> ^id <id1>)
  (flight ^source <stop-over> ^destination <dest> ^cost <c2> ^id <id2>)
  -->
  (make route ^length 2 ^id (gensym) ^traveller <x> ^flight1 <id1>
        ^flight2 <id2> ^cost (compute <c1> + <c2>)))

(p print-lowest-cost-route
  (goal ^name print-route)
  (travel-constraint ^traveller <x> ^hop-number <hops>)
  (route ^length <hops> ^traveller <x> ^cost <c>)
  -(route ^length <hops> ^traveller <x> ^cost <c>)
  -->
  (make min-cost ^traveller <x> ^cost <c> ^length <hops> ^recommended yes))

```

Figure 8.5: PPL productions for airline-route

```

(p two-hops
  (goal ^name compute-routes)
  (traveller ^name <x> ^source <src> ^destination <dest>)
  (flight ^source <src> ^destination <stop-over> ^cost <c1> ^id <id1>)
  (flight ^source <stop-over> ^destination <dest> ^cost <c2> ^id <id2>)
  -->
  (insert (update (make route ^length 2 ^traveller <x> ^flight1 <id1> ^flight2 <id2>)
                  ^cost (compute_costs_2 <c1> <c2>)
                  ^id (vector_gensym (cardinality <id1>) * (cardinality <id2>))))))

```

Figure 8.6: COPL productions for airline-route

8.2 Design of the experiments

These experiments measure and compare the performance of PPL and COPL versions of the benchmark programs. Both versions were compiled at the highest level of optimization available in the respective compilers. The C code generated by the compilers as well as the code for the run-time libraries was compiled using the MIPS *cc* compiler *version 1.31* with the *-O* option. For these experiments, the uniprocessor version of *pplc* was used.

A sequence of experiments was conducted for both versions of each program. Data sets for these experiments were generated by assigning larger and larger values to the data set parameters. Each sequence was terminated when the execution time for either version increased beyond half an hour. Execution time for each experiment was determined using the *time* facility in *csh*. In all cases, it was the PPL version that was first to run out of time. These experiments were conducted on a Decstation 5000/200 with 96 megabytes of main memory running the Mach 2.6 operating system. To further probe the scalability of Collection Rete, additional experiments were conducted for the COPL versions. Most of these experiments were conducted on Decstation 5000/200s with 96 meg memory running Mach 2.6. A few experiments that required a very large amount of memory were conducted on a Decstation 5000/260 with 480 megabytes of main memory running Ultrix 4.3.

8.3 Comparison between PPL and COPL

Figures 8.7, 8.9, and 8.11 compare the growth of end-to-end uniprocessor execution time for PPL and COPL versions of the benchmark programs. For all three programs, both versions take comparable time for small data sets. For larger data sets, the time taken by the PPL version grows much faster than the time taken by the COPL version. Furthermore, ratio of the execution times also increases with the data set size.

Figures 8.8, 8.10, and 8.12 compare how the space needed for match operations grows for the two versions. For all three programs, the match space required by COPL is significantly less than the match space required by PPL and the difference increases with an increase in the data set size. This shows that the performance improvement achieved by COPL is not a time-space tradeoff.

8.3.1 Analysis

For all the three benchmark programs, the COPL version outperforms the PPL version. However, there is a very large variation in the magnitude of the performance difference. On one extreme, the COPL version of *make-teams* is up to 6500 times faster and uses up to 12 times

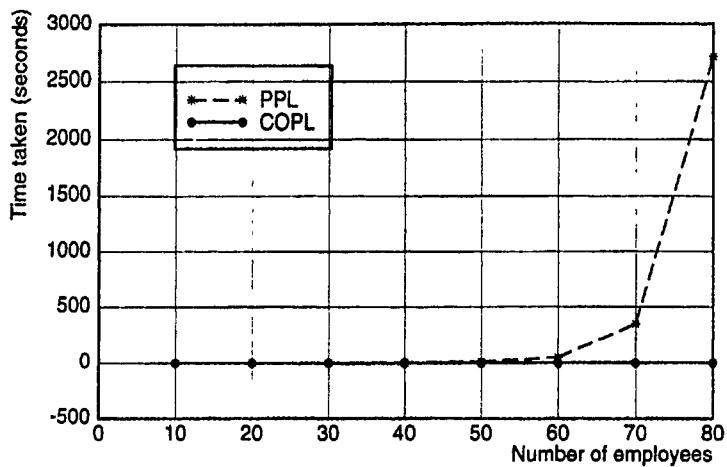


Figure 8.7: Execution time for make-teams

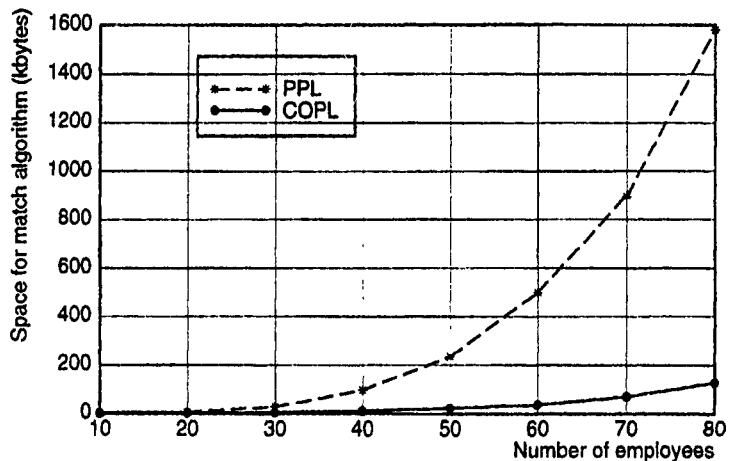


Figure 8.8: Match space for make-teams

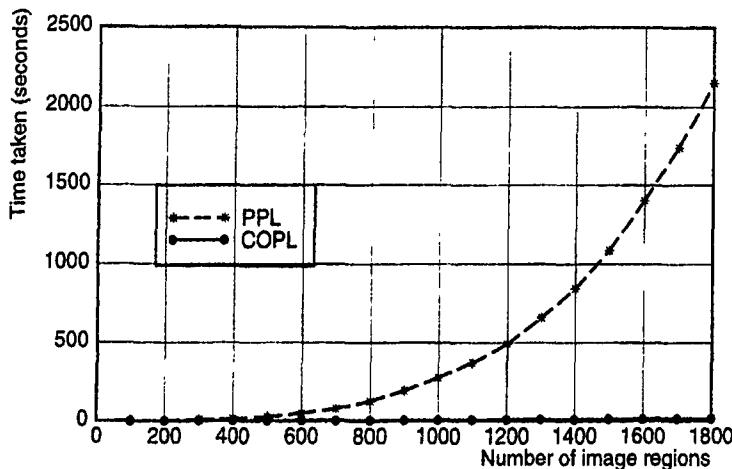


Figure 8.9: Execution time for clusters

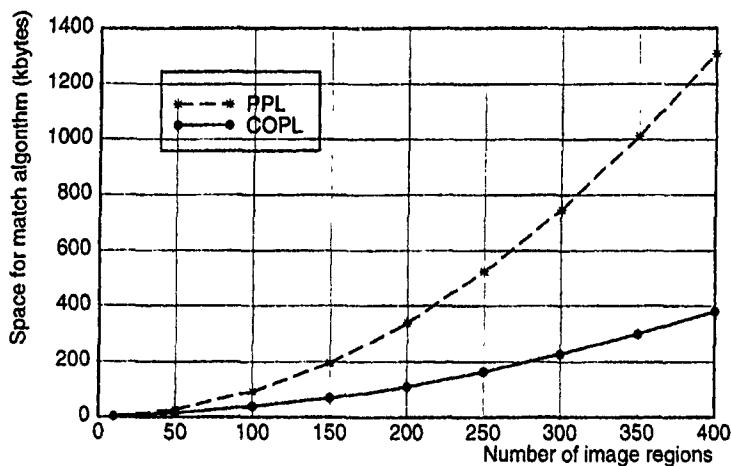


Figure 8.10: Match space for clusters

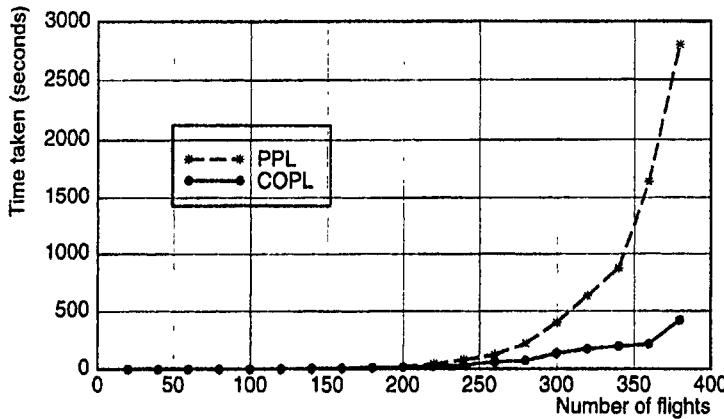


Figure 8.11: Execution time for airline-route

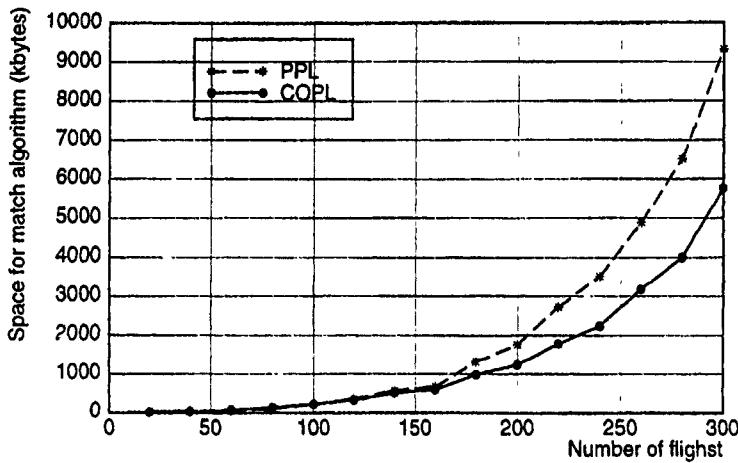


Figure 8.12: Match space for airline-route

less space than the PPL version, on the other extreme, the COPL version of `airline-route` is no more than 7.5 times faster than the PPL version and uses only 1.6 times less space; `clusters` falls in between these extremes with up to 150 fold improvement in execution time and up to 3-4 fold reduction in match space. This section analyzes individual benchmarks to explain this large variation. Since the implementations of PPL and COPL differ only in the support for aggregate operations, the difference in performance can be attributed to the productions that match and operate on aggregates.

make-teams: Only three productions in this program have conditions that match multiple tuples – the production that creates the teams, the production that selects “good” teams and the production that counts the selected teams. COPL versions of these productions are shown in Figure 8.2.

Since the number of compiler experts is limited to five³ and they are equally distributed, each compiler expert has a different value of `previous-project`. All the other employees are randomly, and approximately equally, distributed over all the ten projects. Therefore, five collection-oriented instantiations are generated for the production that creates the teams. In comparison, $O(n^3)$ tuple-oriented instantiations are generated for this production (n being the number of employees).

The “goodness” scores for all the employees are randomly assigned from the range [0,4] and the threshold for a “good” team is 8. There are $5^4 = 625$ ways in which members of a team can be assigned numeric evaluations. Only 54 of these add up to a “goodness” score over 8. Since the numeric evaluations are randomly assigned, the expected value of the fraction of all possible teams that are selected by the second production is $54/625 = 0.0864$. Since a separate tuple-oriented instantiation is generated for every selected team, $O(n^3)$ such instantiations are generated. In comparison, only one collection-oriented instantiation is generated.

The third production counts the number of “good” teams. Under tuple-oriented semantics, a quadratic number of instantiations are generated for counting. Since there are $O(n^3)$ teams, $O(n^6)$ tuple-oriented instantiations are generated. In comparison, only one collection-oriented instantiation is generated.

clusters: There are four productions in this program with conditions that match multiple tuples – the production that computes the distances, the production that creates the clusters, the production that computes the size of individual clusters, and the production that sums the sizes of the clusters. COPL versions of these productions are shown in Figure 8.4.

Since the ratio of the number of seed regions and the number of non-seed regions is fixed (ten), the number of tuple-oriented instantiations of the production that computes the distances is $O(n^2)$, where n is the number of seed regions. In comparison, only one collection-oriented instantiation is generated

³Compiler experts being worth their weight in gold :-)

Since the seed regions are uniformly distributed over the 100x100 grid, their density is $n/10^4$. The expected number of seed regions within $\sqrt{800}$ units of distance from any point is $\sqrt{800}n/10^4$. This is the expected number of clusters that each region occurs in. This is also the expected number of tuple-oriented instantiations of the cluster-creation production generated for every image regions. Since there are $O(n)$ image regions, $O(n^2)$ such instantiations are generated. In comparison, n collection-oriented instantiations are generated, one instantiation per seed-region.

The density of the non-seed image regions is $10n/10^4 = n/10^3$. Therefore, the expected value of cluster size is $\sqrt{800}n/10^3$. Since, counting generates a quadratic number of tuple-oriented instantiations, $O(n^2)$ such instantiations of the third production are generated. In comparison, n collection-oriented instantiations are generated, one per cluster.

Since there are n clusters, and since an accumulation loop generates quadratic number of tuple-oriented instantiations, $O(n^2)$ such instantiations of the fourth production are generated. In comparison, only one collection-oriented instantiation is generated.

airline-route: The only productions whose conditions match multiple tuples are the productions that compute the routes. There are five such productions, for routes of length one through five. There are twenty airports, ten of which are randomly chosen to be hubs. The data set is parameterized by the number of flights from each hub. A separate tuple-oriented instantiation of the k^{th} production is generated for every sequence of flights of length k which originates at the source and terminates at the destination. Consider the simplest case of direct flights. Since either or both of the end-points can be hubs and since there is a 50% probability of any airport being a hub and since the expected number of flights from any hub to any other airport is $n/19$, the expected number of direct flights from the source to the destination is $2 \times 1/2 \times n/19 = n/19$. Therefore, $O(n)$ tuple-oriented instantiations are generated for the first production. Next, consider the case of two-hop flights. Since the intermediate point must be different from both the end-points, the expected number of flight-sequences is $18n/19 \times n/19 = 18n^2/19$. By generalization, $O(n^k)$ tuple-oriented instantiations are generated for the production that computes the routes of length k .

For the production that computes the direct flights, only one collection-oriented instantiation is generated. For the production that computes the two-hop flights, a separate collection-oriented instantiation is generated for every intermediate point. There are 18 such points possible, though not all of them might be linked to both source and destination. In general, C^k collection-oriented instantiations are generated for the production that computes k -hop flights, C being a constant less than 20.

Summary: The difference in the number of instantiations is greatest for `make-teams`. Accordingly, it demonstrates the largest difference in performance. The difference in the number of instantiations is smaller for `clusters` and so is the difference in the performance. The primary reason for the relatively small difference in the performance of the PPL and COPL

versions of `airline-route` is the fact that the data sets used in the experiments correspond to small values of the data set parameter. Since each increment in the data set parameter corresponds to 20 flights, the largest data set shown in the figures corresponds to a parameter value of 19. The larger data sets already show significant divergence in the performance. For sufficiently large data sets, the difference should be comparable to that achieved by `make-teams`. While COPL uses less match space than PPL for all three benchmarks, the reduction in space usage is much less than the reduction in the execution time. There are two reasons for this. First, a large fraction of the space is consumed by right memory nodes. Collection-oriented match does not reduce the cardinality of right memory nodes. Second, collection-oriented match has a space overhead – primarily for the maintenance of equivalence classes and collections. If the average size of collections is large, these overheads are insignificant compared to the space savings but if the average size of collections is small, this overhead can become significant. Therefore, the ratio of the match space used is governed by the number of instantiations and tokens a tuple appears in and by the average size of collections.

8.3.2 Critique

The magnitude of the results presented in the previous subsections raise questions about their fairness and generality. This section discusses these questions.

Is the baseline efficient? The baseline used in these experiments, `pp1c` is a highly optimized implementation. Section 3.4 describes the suite of optimizations that have been incorporated into `pp1c`. Comparison of PPL with CParaOPS5, a compiler previously considered to be state-of-the-art, shows PPL to be between 2 and 90 times faster and using between 2.2 and 4.2 times less space (see Section 4.2). On the other hand, the COPL implementation is a first-cut implementation and is relatively under-optimized. A host of optimizations have been discussed and proposed but are yet to be implemented.

- *Sharing of collections between tokens:* Currently, collections in a parent token are copied whenever a successor is created. This is usually not necessary and it is possible to statically determine when it and when it is not needed. For large collections, avoiding copying collections could make a significant difference.
- *Using equivalence classes as collections:* For productions that contain no negated conditions, it is possible to use the equivalence classes directly as collections. This can reduce the cost of matching a new tuple to the cost of finding the appropriate equivalence class and adding the tuple to it. This optimization avoids performing β tests for tuples that join pre-existing equivalence classes. Since β tests are by far the most expensive part of the match process, this optimization has potential for large speedups for the programs it is applicable to. With the availability of collection-oriented operations, negated conditions are expected to be rare (see Section 8.5.2 for details).

- *Hashing right memories:* As the number of equivalence classes grows, it is beneficial to organize a right memory into a hash table of equivalence classes.

For small tuple-spaces, the performance of PPL is comparable with that of COPL. In fact, for small tuple-spaces of airline-route, the PPL version is up to 2.1 times faster than the COPL version. Furthermore, as the results indicate, PPL has been able to deal with tuple-spaces whose maximum size is over 139,000 tuples. No other production system implementation has been reported to be able to process such large tuple-spaces.

How general are the results? These experiments study the performance of two match algorithms, one tuple-oriented and the other collection-oriented, in the presence of a combinatorial explosion in the number of instantiations and tokens. They show that collection-oriented algorithms are better able to deal with this combinatorial explosion, the magnitude of the difference in performance depending on the selectivity of the tests in the productions and the patterns in the data set. Programs processing large tuple-spaces have been used as benchmarks but the occurrence of a combinatorial explosion is not limited to such programs. Programs processing much smaller data sets can experience a combinatorial explosion in the number of instantiations if the selectivity of the tests is low. For example, most Soar [66] programs process relatively small tuple-spaces. However, the learning procedure sometimes creates productions of low selectivity, called *expensive chunks*. These productions are so expensive to match that they cause Soar to slowdown after learning instead of speeding up.

Are the data sets unrealistic? The restrictions placed on two of the data sets, *make-teams* and *clusters* may appear to be unrealistic but they have been imposed to *reduce* the difference between the performance of PPL and COPL. For example, the restriction of five compiler experts in *make-teams* limits the number of instantiations for the production that generates the teams to $O(n^3)$ and for the production that counts "good" teams to $O(n^6)$. In the absence of this restriction, $O(n^4)$ tuple-oriented instantiations would be generated for the former and $O(n^8)$ tuple-oriented instantiations for the latter.

8.4 Scalability of collection-oriented match

For a collection-oriented match algorithm, the number of instantiations for a production depends on the extent to which the tuples matching individual conditions can be grouped together. If all the tuples matching every condition can be grouped, only one instantiation is generated for every production. On the other hand, if no grouping is possible, a collection-oriented algorithm reduces to its tuple-oriented analogue. In other words, the number of instantiations for a production depends on how the collections of tuples corresponding to each condition are partitioned, or *fragmented*.

Fragmentation: Consider the production and the tuple-space in Figure 8.13. This production matches pairs of objects of the same type. Since all the objects in the tuple-space are of the same type, the tests in the production are unable to distinguish between them and only one collection-oriented instantiation is generated – $\langle \{t1, t2, t3, t4\}, \{t1, t2, t3, t4\} \rangle$.

```
(p pairs
  (object ^id <object1> ^type <>)
  (object ^id <object2> ^type <>)
  -->
  (make pair ^first <object1> ^second <object2>))

t1: (object ^id 1 ^type box ^size large)
t2: (object ^id 2 ^type box ^size large)
t3: (object ^id 3 ^type box ^size small)
t4: (object ^id 4 ^type box ^size small)
```

Figure 8.13: Example production and tuple-space

Now suppose the tests in the production are changed so that the size of the objects is also tested. For example, if the test `^size <size>` is added to both the conditions. The modified production matches pairs of objects of the same type and the same size. Since the objects are of different sizes, the modified tests are able to distinguish between them. The original instantiation is fragmented into two – one with large objects, $\langle \{t1, t2\}, \{t1, t2\} \rangle$, and the other with small objects, $\langle \{t3, t4\}, \{t3, t4\} \rangle$.

On the hand, suppose the production remains as it is but the type of the second and fourth objects is changed to `circle`. In this case, the tuple-space would be:

```
t1: (object ^id 1 ^type box ^size large)
t2: (object ^id 2 ^type circle ^size large)
t3: (object ^id 3 ^type box ^size small)
t4: (object ^id 4 ^type circle ^size small)
```

Since the objects are no longer of the same type, tests in the original production are able to distinguish between them. The original instantiation is fragmented into two – one with box objects, $\langle \{t1, t3\}, \{t1, t3\} \rangle$, and the other with circle objects, $\langle \{t2, t4\}, \{t2, t4\} \rangle$. Now, if the additional test for the size of objects is included, each of these fragment undergo fission resulting in the generation of four instantiations – $\langle \{t1\}, \{t1\} \rangle$, $\langle \{t2\}, \{t2\} \rangle$, $\langle \{t3\}, \{t3\} \rangle$ and $\langle \{t4\}, \{t4\} \rangle$.

As the above examples show, fragmentation occurs when the tests in the production(s) are able to distinguish between different tuples. As a program evolves, this can be due to an increase in the specificity of the tests or a change in the distribution of values in the data. As the data set for a given program increases, the number of instantiations and tokens is likely to remain the same if the additional tuples have the same value(s) for the fields being tested as existing tuples. For example, by adding

```
t5: (object ^id 5 ^type box ^size large)
t6: (object ^id 6 ^type box ^size small)
```

to the tuple-space in Figure 8.13, no additional instantiations or tokens are generated. On the other hand, if the tuples being added to a growing data set have different value(s) for the tested fields as existing tuples. This leads to the generation of additional instantiations and tokens. For example, adding

```
t5: (object ^id 5 ^type circle ^size large)
t6: (object ^id 6 ^type circle ^size small)
```

to the tuple-space in Figure 8.13, would lead to the generation of an additional instantiation – $\langle (t5), (t6) \rangle$.

As can be seen from the above examples, the degree of fragmentation depends on the specificity of the tests in the productions *relative* to a given tuple-space.

If the performance of a collection-oriented match algorithm is to scale with tuple-space size, the rate at which new partitions are generated should be much smaller than the rate at which tuples are added to the tuple-space. In the ideal case, addition of tuples does not increase the fragmentation. For example, if all the tuples being added to the tuple-space in Figure 8.13 correspond to large boxes. In the worst case, every new tuple added increases the fragmentation. For example, if every pair of tuples added to the tuple-space in Figure 8.13 corresponded to objects of a unique type.

The following section examines the scalability of Collection Rete, as an exemplar collection-oriented match algorithm. It provides an empirical demonstration of how the interaction between the specificity of the productions and the data distribution governs the scalability of collection-oriented match algorithms.

8.4.1 Scalability of Collection Rete

This section examines the scalability of Collection Rete based programs in greater detail. It uses the number of tuple-space modifications per second as the metric. This metric is computed by dividing the total number of tuple-space modifications by the total execution time and is referred to as the *tuple processing rate*. Since match algorithms are incremental and

process modifications to the tuple space rather than the tuple-space itself, the total number of modifications is a fairer measure of the amount of work than the number of tuples in the initial tuple space. In the best scenario for scalability, the tuple processing rate remains constant (or increases) as the data set size is increased. A constant tuple processing rate indicates that the cost of processing a tuple does not grow with tuple space size and that there is no combinatorial explosion in the number of instantiations and tokens. On the other hand, if there is a combinatorial growth in the number of instantiations and tokens as the tuple space size increases, the average time needed to process a single tuple goes up resulting in a lower tuple processing rate.

make-teams: Increasing the data set size for `make-teams` does not lead to an increase in the fragmentation since all the new tuples are added to existing tuples. As a result, the tuple processing rate for `make-teams` remains within a relatively small range, 26,000 tuples/second to 32,000 tuples/second, over a two orders of magnitude increase in the tuple-space size. Figure 8.14 shows how the tuple processing rate varies with tuple-space size. Note that the graph is plotted on a semi-log scale. The rate drops for very large tuple-spaces. This drop can be attributed to paging effects as the total memory required for these cases is close to the physical memory available on the machine (96 megabytes).

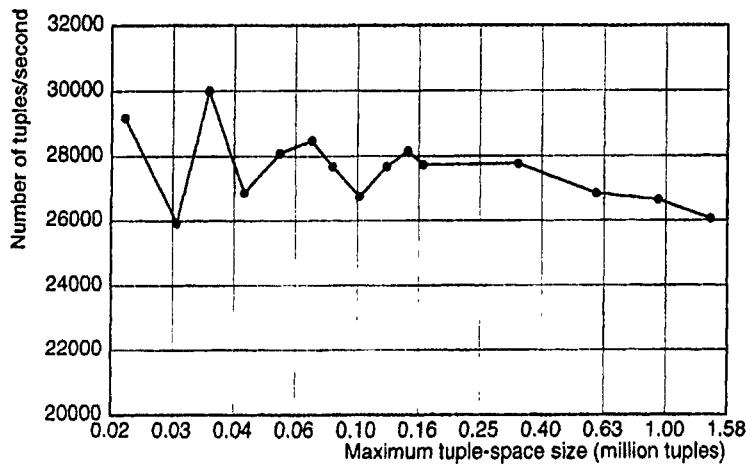


Figure 8.14: Tuple processing rate for `make-teams`

clusters: Increasing the data set size for `clusters` does not increase the number of instantiations for the production that creates the teams and the production that accumulates the sizes of

the clusters. New instantiations are generated for the productions that create clusters and count the number of their members. However, the rate at which new instantiations are generated is much smaller than the rate at which tuples are added (the average size of clusters ranges from 29 to 395 and increases with an increase in data set size). Figure 8.15 shows that the tuple processing rate for clusters increases with tuple-space size. Note that the graph is plotted on a semi-log scale. Like make-teams, the tuple processing rate for clusters drops for very large tuple-spaces due to paging effects.

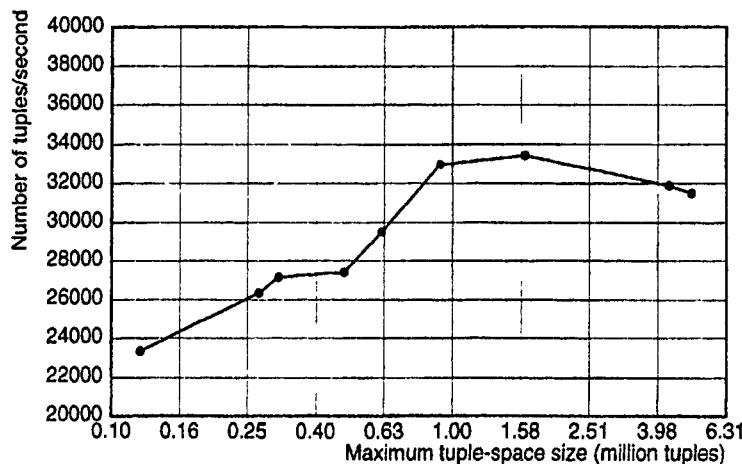


Figure 8.15: Tuple processing rate for clusters

airline-route: The key productions for this benchmark are the productions that find routes of different lengths from the source to the destination. One instantiation of these productions is generated for every route found and corresponds to all the flight sequences over this route. Since increasing the data set size for **airline-route** consists of populating a random graph, the number of routes does not increase smoothly with an increase in number of flights. On one hand, addition of a single flight may not contribute towards any route and on the other, it may provide the vital link between two hitherto independent subgraphs. Figure 8.16 illustrates this. Assume the solid edges are the existing flights and the dashed edges are two flights that has just been added. Flight A does not lie on any route from the source to the destination, whereas addition of flight B creates $4 \times 4 = 16$ new routes. As a result, the number of instantiations increases by spurts. Accordingly, the tuple processing rate for **airline-route** varies spasmodically with tuple-space size. Figure 8.17 plot the tuple processing rate against tuple-space size. The

low tuple processing rate is mainly due to the low density of the flight graph. As the density of the flight graph increases, the likelihood of a new flight being along an existing route increases. As a result, the rate at which new instantiations are generated will decrease as the data set size grows.

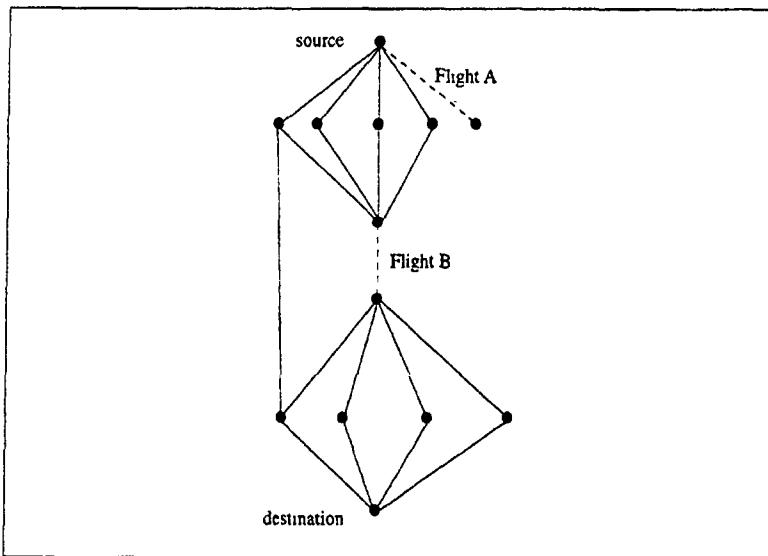


Figure 8.16: Addition of new flights to existing flight database

Finally, Table 8.1 contains information about the largest data set processed by each of the benchmark programs. It shows that the COPL version of *clusters* was able to process over 10 million tuples in less than four minutes on a Decstation 5000/260 with 480 megabytes of main memory.

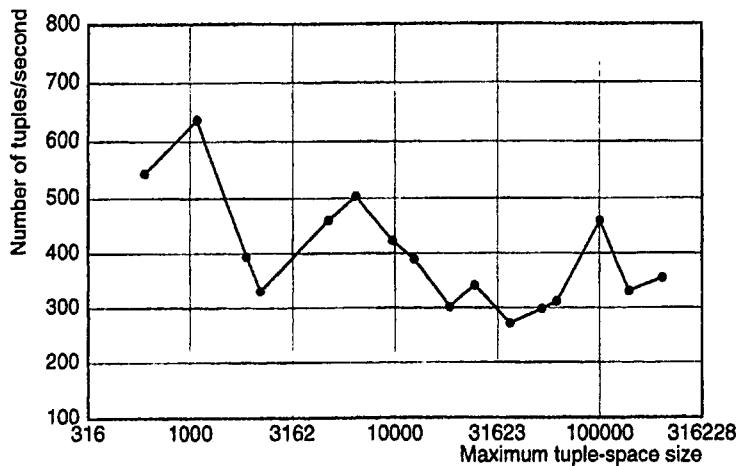


Figure 8.17: Tuple processing rate for airline-route

Program	Data set size	execution time	match space	number of tuples
make-teams	400 employees	53.5s	21113K	1393457
clusters	10000 regions	205.9s	91925K	10165576
airline-route	400 flights	574.5s	2733K	203881

Table 8.1: Largest experiments

8.5 Observations

8.5.1 Relational match tests are inefficient

A frequently used idiom in production system programming is the use of relational tests in the conditions to extract the minimum (or maximum) element of a collection of values. Figure 8.18 shows an example. As each `data-item` tuple is added to the tuple-space, it is compared with all `data-item` tuples already in the tuple-space. This results in $O(n^2)$ comparisons, where n is the number of tuples. Depending on the order in which the tuples are inserted, this also leads to the generation and deletion of $O(n)$ instantiations. If the tuple with the largest value is inserted first, only one instantiation is generated whereas if the tuples are inserted in increasing

order of value, a new instantiation is generated for every tuple.

```
(p find-min
  (data-item ^id <id> ^value <v>)
  -(data-item ^id <> <id> ^value <>)
  -->
  (write "The minimum value is " <v>))
```

Figure 8.18: Finding the minimum element using relational tests

A more efficient way of extracting the minimum element from a collection is illustrated in Figure 8.19. This production extracts all the data-item tuples and calls a C routine, `find_min()`, to extract the minimum element. This routine scans the collection of values to find the minimum element and needs only $O(n)$ comparisons.

```
(p find-min-new
  (data-item ^value <v>)
  -->
  (write "The minimum value is " (find_min <v>)))

int find_min(seq)
sequence seq;
{ /* the sequence is guaranteed to have one element, else no instantiation */
  int min = value(seq);

  for (seq = next(seq), !empty(seq); seq = next(seq))
    if (value(seq) < min) min = value(seq);

  return(min);
}
```

Figure 8.19: Finding the minimum element using a procedure

8.5.2 Negation needed infrequently

Negated conditions are used in several commonly used production system programming idioms including loop termination, aggregate updates, extraction of minimum (or maximum) element. Negated conditions are used in these cases for their ability to search the entire tuple-space and to make universally quantified assertions like *there is no tuple whose data field has a value greater than 50*. Figure 8.20 shows examples of the first two idioms. Figure 8.18 shows an example of finding the minimum element. Such idiomatic use accounts for most of the uses of negated conditions, testing for the non-existence of a particular value is relatively rare.

```

(p loop
  (goal ^name accumulate-values)
  (accumulated-sum ^value <sum>)
  (data-item ^value <v>)
  -->
  (remove 3)
  (modify 2 ^value (compute <sum> + <v>)))

(p loop-termination
  (goal ^name accumulate-values)
  (accumulated-sum ^value <sum>)
  -(data-item)
  -->
  (write "The sum is " <sum>))

(p aggregate-update
  (region ^type seed ^id <seed> ^x <x1> ^y <y1>)
  (region ^type non-seed ^id <member> ^x <x2> ^y <y2>)
  -(distance ^from <seed> ^to <member>)
  -->
  (make distance ^from <seed> ^to <member>
    ^value (distance <x1> <y1> <x2> <y2>)))

```

Figure 8.20: Programming idioms using negation

Collection-oriented production languages allow non-negated conditions to match the entire

tuple-space. Therefore, it reduces the need for the use of negated conditions. Figure 8.21 shows how each of the three idioms mentioned above can be implemented in COPL. COPL versions of the benchmarks used in these experiments contained no negated conditions. In comparison, the OPS5 versions needed quite a few negated conditions and the PPL versions needed a small number of them.

Match algorithms, including Collection Rete, can be optimized to take advantage of the absence of negated conditions. In a production that has no negated conditions, like the production that creates the teams in make-teams, it is not necessary to create a list of pointers to represent the collection corresponding to individual conditions. Instead, it is possible to use pointers to equivalence classes directly. In such cases, processing a new tuple can often be reduced to finding the appropriate equivalence class and adding the tuple to it.

```
(p loop-no-termination-needed
  (goal ^name accumulate-values)
  (data-item ^value <v>)
  -->
  (make accumulated-sum ^value (sum <v>)))

(p aggregate-update
  (region ^type seed ^id <seed> ^x <x1> ^y <y1>)
  (region ^type non-seed ^id <member> ^x <x2> ^y <y2>)
  -->
  (insert (update (make distance ^from <seed> ^to <member>
    ^value (all_distances <x1> <y1> <x2> <y2>))))
```

Figure 8.21: Programming idioms rewritten in COPL

8.5.3 Condition ordering less important for efficiency

Reordering conditions is a widely used optimization in production system programs. The basic idea is to limit the number of intermediate results by ordering the conditions of a production in decreasing order of selectivity. Consider the production and the tuple-space in Figure 8.22. With the given order, there are six tuple pairs that match the first two conditions, $\langle t1, t4 \rangle$, $\langle t1, t5 \rangle$, $\langle t2, t4 \rangle$, $\langle t2, t5 \rangle$, $\langle t3, t4 \rangle$ and $\langle t3, t5 \rangle$. Since the last condition matches a single tuple, there are six tuple triples that match the entire production. The total number of

tuple combinations generated is 12. Now, if the order of the conditions is reversed, only two tuple pairs match the first two conditions, $\langle t_6, t_4 \rangle$ and $\langle t_6, t_5 \rangle$. Since, the third condition matches three tuples, six tuple-oriented instantiations are generated. The total number of tuple combinations, in this case is 8.

```
(p reordering-example
  (spy ^name <spy>)
  (knaver ^name <knaver>)
  (knight ^name <knight>)
  -->
  (make ^first <spy> ^second <knaver> ^third <knight>))

  t1: (spy ^name Mata-Han)
  t2: (spy ^name Sergie)
  t3: (spy ^name Powell)
  t4: (knaver ^name Guy-Fawkes)
  t5: (knaver ^name Petain)
  t6: (knight ^name Lancelot)
```

Figure 8.22: Example illustrating the reordering optimization

Since collection-oriented match avoids creating explicit cross-products as far as possible, the order of conditions is not as important. In the above example, a collection-oriented match algorithm generates only one pair of collections matching the first two conditions and only one collection-oriented instantiation for the production irrespective of the condition ordering. In general, the importance of condition ordering in collection-oriented match algorithms depends on the degree of fragmentation. Accordingly, condition ordering makes no difference in the number of instantiations and tokens generated for `make-teams` and `clusters`, whereas for `airline-route`, different orderings result in the generation of different numbers of tokens.

8.5.4 Cardinality of variable values depend on condition ordering

In a tuple-oriented instantiation, each variable is bound to exactly one value, this value being the same for all condition ordering. In a collection-oriented instantiation, each variable is bound to a collection. If a variable occurs in multiple conditions, the collection bound to the variable is created by extracting values from the tuples matching the first condition it occurs

in. For example, the collection bound to the variable `<num>` in Figure 8.23 consists of the values of the `^num-of-classes` field of `student` tuples matching the first condition. If the condition matching juniors occurs first, cardinality of `<v>` is four, otherwise it is one. To avoid potential errors relating to this phenomenon, the binding occurrence of a variable should be syntactically differentiated from rest of its occurrences.

```
(p cardinality-example
  (student ^year junior ^num-of-classes <num>)
  (student ^year sophomore ^num-of-classes <num>)
  ->
  (write "Number of matched pairs of students: " (cardinality <num>)))

  t1: (student ^id 1 ^year junior ^num-of-classes 4)
  t2 (student ^id 2 ^year junior ^num-of-classes 4)
  t3: (student ^id 3 ^year junior ^num-of-classes 4)
  t4: (student ^id 4 ^year junior ^num-of-classes 4)
  t5. (student ^id 5 ^year sophomore ^num-of-classes 4)
```

Figure 8.23: Example illustrating difference in cardinality of variable values

8.5.5 Interaction with parallelism

Chapter 5 identified three major limitations on parallelism in production system programs written in tuple-oriented languages. First, the small average task size in efficient production system programs results in a high parallelization overhead. Second, the cost of matching after every instantiation firing increases the cost of sequential loops. This limits the achievable speedup as an Amdahl's law effect. Third, occurrence of cross-products high in the Rete network limits the average number of tasks available and leads to low processor utilization.

Collection-oriented match is able to eliminate or alleviate all three limitations. Collection-oriented tokens are essentially a grouping of a collection of tuple-oriented tokens. To process a collection-oriented token, it is necessary to compare a sequence of collections of tuples (the token) with a collection of equivalence classes (the right memory). In comparison, processing a tuple-oriented token consists of the comparison of a sequence of tuples (the token) with a collection of tuples (the right memory). Depending on the degree of fragmentation, collection-oriented match is able to group together arbitrarily large number of tuple-oriented tokens.

As illustrated in Figures 8.2, 8.4 and 8.21, collection-oriented production languages make it possible to move sequential loops from the production system paradigm, where they are inefficient, to procedural languages, where they can be efficiently implemented. Furthermore, many of these operations can be parallelized in a procedural paradigm. For example, the accumulation loop in Figure 8.3 can be replaced by a parallel tree-based addition algorithm which takes $O(\log n)$ time.

Since collection-oriented match delays the generation of cross-products, it avoids the bottleneck that arises due to the use of sequencing tuples in some tuple-oriented programs, for example, *waltz*, one of the benchmarks used in the parallelism experiments (see Section 5.2.3). By generating a single successor relatively quickly, collection-oriented match avoids the bottleneck.

8.5.6 Less restrictive is better

One of the primary optimizations used by production system programmers to tune their programs is to increase the restrictiveness of the conditions. Programming texts devote several pages to various ways of limiting the number of tuples that match individual conditions. For example, see [11][pages 243-59]. However, as the size of the tuple space grows, increasing the restrictiveness of the tests usually increases number of instantiations and thereby leads to a degradation in performance. Section 8.4 illustrates this with an example. Figure 8.24 shows how a decrease in restrictiveness reduces the number of instantiations. This production pairs each data-item with the collection of data-items whose value is less than its own. For the given tuple-space, four collection-oriented instantiations are generated, one each for t_1, t_2, t_3 and t_4 . The second production eliminates the test on the value of the data-items and uses a procedure call to create the threshold partitions. Only one collection-oriented instantiation is generated for this production.

8.5.7 Programming guidelines

This section presents some guidelines for programming collection-oriented production languages.

Move loops to procedural languages: As mentioned several times in this dissertation, the production system paradigm is not suitable for implementing loops, in particular loops with inter-iteration dependencies. COPL provides collection-oriented tuple-space operations to implement loops that modify the tuple-space. Other loops should be implemented by using match to extract the data items to be processed and by calling a procedural language routine to actually perform the computation. Examples of such loops can be found in Figures 8.2, 8.4, 8.6, 8.19, 8.21 and 8.24.

```

(p threshold-1
  (data-item ^id <threshold> ^value <v>)
  (data-item ^id <items> ^value <<v>>)
  -->
  (some action <items>))

(p threshold-2
  (data-item ^id <thresholds> ^value <v1>)
  (data-item ^id <items> ^value <v2>)
  -->
  (some action (threshold <thresholds> <items>)))

t1: (data-item ^id 1 ^value 5)
t2: (data-item ^id 2 ^value 4)
t3: (data-item ^id 3 ^value 3)
t4: (data-item ^id 4 ^value 2)
t5: (data-item ^id 5 ^value 1)

```

Figure 8.24: Decreasing restrictiveness reduces number of instantiations

Avoid selection via the conflict set: A programming idiom used commonly in production systems is selection from a set. A separate instantiation is generated for every element and the instantiation selection algorithm is used to select one of the values. The other instantiations are then eliminated from the conflict set. In a collection-oriented production language, this can be implemented by extracting all the elements and using a selection predicate in the actions. This allows a more efficient implementation of selection as well as flexibility in selection predicates. An example is shown below.

```

(p selection
  (data-item ^value <v>)
  -->
  (make selected-item ^value (selection_filter <v>)))

```

Use match only for retrieval: Most production system languages allow relational tests in the conditions, some even allow arbitrary user-written tests. The goal of including such tests in the language is to allow programmers to use the match procedure for operations other than

retrieval. For example, a relational test between values in different tuples can be used to extract the minimum (or maximum) element from a collection of values (see Figure 8.18). Figure 8.24 shows how relational tests can be used selecting values from a collection ("select all values less than a threshold"). In many cases, the use of such tests forces the creation of a large number of instantiations. This could be avoided if the match procedure is used only for retrieving values from the tuple-space and other operations are performed by calls to procedural language routines. Figure 8.24 shows an example. Even the operations that do not increase the number of instantiations, like `find-minimum` in Figure 8.18, they are less efficient when embedded in the match procedure. See Section 8.5.1 for an example.

Reduce restrictions: As discussed in Section 8.5.6, reducing the restrictiveness of the conditions often reduces the number of collection-oriented instantiations generated. In many cases, it is possible to perform the selection achieved by the eliminated restrictions by calling a filter procedure. For examples, see Figures 8.19 and 8.24.

8.5.8 It is possible to be even more lazy

Consider the task of finding all nodes in a network that are three hops away from a given node. Figure 8.25 shows a production that implements this. Under tuple-oriented semantics, one instantiation is generated for every path of length three from the root node. For the network in Figure 8.26, ten tuple-oriented instantiations are generated, one each for A and D and three each for B and C. Since the production conditions discriminate between every such path, collection-oriented match would generate the same set of instantiations. However, the operations performed by the production need only the leaf nodes and not the paths to these nodes. Since collection-oriented match guarantees the mutual consistency between all tuples in a collection-oriented instantiation, it is unable to collapse all the paths to a node. A variation of collection-oriented match that relaxes the mutual consistency guarantee would be able to deal with such situations by extending the laziness to generate the paths only if needed. This is subject of future work.

```
(p find-three-deep-leaves
  (root-node ^id <root>)
  (node ^id <intermediate1> ^parent <root>)
  (node ^id <intermediate2> ^parent <intermediate1>)
  (node ^id <leaf> ^parent <intermediate2>)
  -->
  (do-something <leaf>))
```

Figure 8.25: Production that finds leaves two hops away from the root

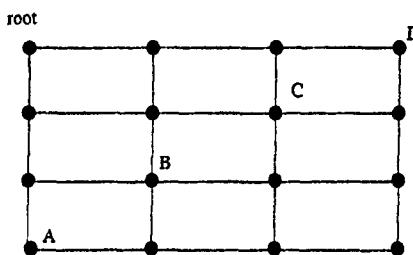


Figure 8.26: Example network

Chapter 9

Related Work

In early production system programs, match was by far the most expensive phase. As a result, initial research on parallelizing production system programs focussed on parallelizing the match phase. Several approaches were investigated usually involving special-purpose hardware. Section 9.1 briefly describes these efforts. These efforts were limited partly by Amdahl's law (since only the match phase was parallelized) and partly by the high parallelization overheads.

The limited success of these projects prompted investigation of various ways to detect which instantiations can be fired in parallel. Most of these efforts extended the analysis proposed by Ishida and Stolfo in their seminal paper[56]. Section 9.2 describes these efforts. As mentioned in Chapter 3, these efforts were limited by the the data-dependent nature of the production system paradigm and the lack of knowledge of about the run-time contents of the tuple-space.

The earliest explicitly parallel production system language proposed was *Herbal*[122] which allowed the programmer to specify which conditions could match multiple tuples and extended the tuple-space operations to process collections of tuples. *Herbal* was not implemented. It was followed by several languages including C5[36], Ariel[44], PARULEL[107] and OPRL[128]. Section 9.3 describes these languages briefly.

Combinatorial explosion in the number of instantiations and partial matches has long been known to be a major efficiency problem for production systems programs. Production system textbooks, for example [11], devote several pages to programming idioms to avoid or reduce cross-products. There are three constraints on the match problem that make it impossible to guarantee the elimination of this combinatorial explosion:

- The if-part of a production consists of a conjunction of conditions,
- Each condition can potentially match the entire tuple-space and

- The match procedure must be sound and complete, that is, it must generate all matched instantiations and must not generate any spurious instantiations.

Researchers seeking to improve the scalability of production system programs have proposed relaxing one or more of these constraints. Section 9.4 describes these approaches. The collection-oriented match approach described in this dissertation is an attempt to tame the combinatorial explosion without relaxing any of the constraints. As such, it is unable to *guarantee* the absence of a combinatorial explosion but, as shown in Chapter 8, it performs quite well for a large class of programs.

9.1 Parallel match

9.1.1 Tree-structured architectures

Several research efforts in the early 1980s attempted to use binary-tree-structured architectures for implementing production system programs. The first of these was the DADO machine[106]. A full-scale version of the DADO machine would comprise of thousands of 8-bit processing elements with 20kbytes of local memory. Two major prototypes were built, the DADO-1 with 15 processing elements and the DADO-2 with 1023 processing elements.

DADO was a partitionable SIMD machine, that is, the full binary-tree could either be run as a single SIMD machine or it could be partitioned into disjoint subtrees each of which ran as an independent SIMD machine. Several algorithms were proposed for implementing production system match on the DADO[37, 77, 105], the most promising being a parallel version of Rete[37] and Treat[77]. These algorithms divide the DADO tree into three levels – the *Upper tree*, which is used for synchronization as well as for the select and the act phases, the *PM level* which is used to perform the β tests and the *WM-subtrees* which are used to perform the α tests and to implement the memory nodes.

The NON-VON machine[51] was similar to DADO in that it used a large number of small processors. The analysis presented in [51] assumes a configuration with 16K processors. Three NON-VON prototypes were built, the largest having 8K processors. The full-scale version of NON-VON was envisioned to have a million processors. Like DADO, NON-VON was a partitionable SIMD machine but the maximum number of partitions were limited by the number of processors in the upper-most layer. NON-VON used 8-bit processors with 64 bytes of local memory. In addition to the binary-tree interconnection network, the processors were also connected by a two-dimensional mesh network. However, this network was not used for production system programs. A parallel version of Rete was used to implement production system programs on the NON-VON. The processors in the binary-tree were used to implement the α tests and implement the memory nodes. The β tests were performed by a more powerful

processor close to the root of the tree and the select and act phases were implemented on the host processor. Due to the small size of the memory associated with individual processors, the contents of individual memory nodes were distributed over several processors, each processor containing at most one token.

The CUPID machine[61] appears to be a modified version of NON-VON with fewer and more powerful processors. It was envisioned to have between 64 to 512 32-bit RISC processors rated at 5-6 MIPS. Like NON-VON, CUPID has two interconnection networks – a binary-tree network for broadcasting data and collecting results and a two-dimensional mesh for local inter-processor communication. A parallel version of Rete, similar to the one used for NON-VON, was used to implement production system programs on CUPID. However, since CUPID contains fewer and more powerful processors, multiple tokens were allocated to each processor. At the end of every match phase, the mesh network was used to balance the load by migrating tokens. Since CUPID had only one class of (powerful) processors, the α and β tests were performed by the same processors that implemented the memory nodes. This is different from NON-VON where the numerous simple processors are used to implement the memory nodes and the α tests and the few powerful processors are used to perform the β tests.

The architecture proposed by Oflazer in his thesis[90] is similar to those above. It consists of 256-1024 processors, rated at 5-10 MIPS, connected by a binary-tree interconnection network with no mesh network as in CUPID or NON-VON. However, the algorithm proposed by Oflazer is significantly different. This algorithm, referred to as Dynamic Join, saves partial matches for not just a particular sequence of conditions but *all* possible sequences. Each production is assigned to a subset of the leaf processors and each processor is responsible for some subset of the productions. Each processor computes and maintains the match state for all productions allotted to it and communicates the instantiations generated to the controller at the root of the binary-tree.

9.1.2 Data-flow architectures

The earliest data-flow architecture proposed for production system programs was the Pesa - 1[102]. It was organized around a sequence of buses. Every processor was connected to two of these buses, it obtained its input from the first bus and placed its output on the second. Figure 9.1 shows an example configuration. Rete was used as the match algorithm for Pesa - 1. The processors in the upper-most layer perform the α tests and implement the right memory nodes. Each subsequent layer of processors is assigned all the nodes at the corresponding depth in the Rete network. They implement both the β tests and the memory nodes. The last two layers implement the select and the act phases, the output of the act phase being placed on the input bus of the topmost layer. The number of layers can be changed to suit the program being implemented.

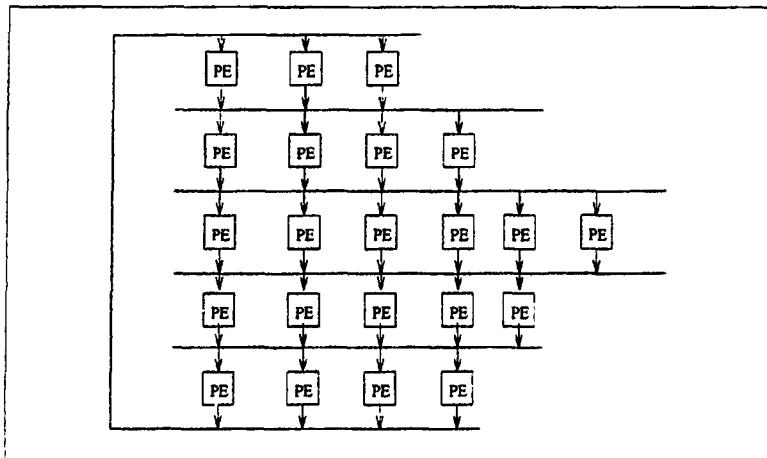


Figure 9.1: The Pesa - 1 architecture

Gaudiot, Lee and Sohn[34] proposed a mapping of Rete onto the MIT tagged-token dataflow architecture[4]. They proposed to use a variation of the Rete network as the dataflow graph. To avoid bottlenecks due to large fanout nodes, which arise in the Rete network due to sharing of common conditions and condition prefixes, they proposed to disable sharing and match each production independently. Their mapping partitions n processors into \sqrt{n} groups. The i^{th} group is assigned the α tests and the right memory node for conditions with i tests. All the β memories are assigned to processors in group 0. Their proposal considers only the match phase and does not make any provision for the select and act phases.

Cheng and Wu[16] contend that the MIT tagged-token architecture is unsuitable for symbolic computation in general and production system programs in particular and propose a dataflow architecture customized for production system execution. This architecture, referred to as DFLOPS, consists of a set of deeply pipelined custom processors, a set of interleaved memory banks and a switching network connecting the two. The instruction set for the processors includes instructions to implement the α and β tests, to create, copy and delete partial matches and to fire instantiations. No prototypes have yet been built.

9.1.3 Shared memory architectures

The earliest shared memory architecture used for implementing production system programs was the C.mmp[71]. The C.mmp consisted of 16 PDP-11 processors connected to a shared memory via a crossbar switch. This work was done prior to the discovery of the incremental match algorithms and recomputed the set of instantiations for every cycle. Since the C.mmp did not have caches, a fair amount of effort was spent in distributing code and data among the different memory modules.

The Production System Machine (PSM) proposed by Anoop Gupta in his thesis[38] consisted of a small number of powerful RISC processors (32-64), a similar number of memory banks and a hardware scheduler connected by a single bus. A small private memory as well as a cache was associated with each processor to reduce the traffic to the shared memory. Gupta argued that the parallelism available in production system programs that had been studied was small and that 32-34 processors should be enough. The hardware scheduler was necessitated by the fine-grain of the individual match tasks (200-800 instructions). The shared bus interconnection (as opposed to an omega network or a shuffle exchange network) was proposed to facilitate solution of the cache-coherency problem. A parallel version of Rete was used to implement production system programs on PSM. Details of this algorithm can be found in Section 2.5. No prototype of PSM was built. However, the design of PSM was used for an implementation of OPSS on the Encore Multimax[41]. The Encore Multimax, with its 16 processors, large shared memory, fast bus and snooping caches was a good approximation to PSM except that the scheduler had to be implemented in software.

Another shared memory architecture proposed for production system execution was the MANJI machine[83]. MANJI consisted of tens of 32-bit processors connected to a shared memory via a single bus. In addition to the bus, MANJI provided a multicast mechanism. MANJI processors had no cache. A parallel version of Rete was used to implement production system programs on MANJI. This version of Rete considers the nodes of the Rete network as a set of interconnected objects passing partial and complete matches as messages. These nodes are statically partitioned among the processors. The multicast mechanism is used to communicate partial and complete matches between processors.

9.1.4 SIMD architectures

The earliest attempt to use SIMD architectures for production system execution was Forgy's mapping of Rete onto the ILLIAC-IV[25]. In this mapping, the productions are divided into 64 partitions, the number of processors in ILLIAC-IV. Separate Rete networks are created for the productions in each partition. All the processors execute the Rete algorithm in lock-step. For example, all processors will perform the α tests before any of them performs a β test.

The *Concurrent Inference System (CIS)*[7] implements production system programs on the Thinking Machines CM-1, a massively parallel SIMD machine with 64K simple processors. CIS takes the production system and a description of all the values that can be bound to each of its variables and compiles them into an *activity flow network*, a static network of simple thresholded computational devices. CIS side-steps the issues of binding arbitrary values to a variable and of creating an arbitrary number of instantiations of productions, the argument being many practical production system programs do not need these features (for example, Mycin[19]).

9.1.5 Message-passing multicomputers

Gupta and Tambe[43] examined the suitability of low-latency fine-grain message-passing machines for production system execution. They proposed a fine-grain mapping of Rete onto a group of message-passing processors. A small number of processors are assigned for α tests and the select and act phases; majority of the processors are used to implement the memory nodes and to perform the β tests. The memory nodes are implemented by a pair of distributed hash tables – one for all the right memory nodes and the other for all the left memory nodes. Each hash bucket is assigned to one of the processors. The number of processors is assumed to be large enough to allow one processor to be assigned to each hash bucket.

Acharya *et al.*[2] examined low-latency medium-grain message-passing machines. Their mapping is a variation of Gupta and Tambe's. In this mapping, there are no dedicated processors for α tests. Instead, all the match processors perform α tests prior to performing memory node operations or β tests. Since this mapping assumes fewer processors (16-64) with larger private memories, several hash buckets are assigned to each processor.

9.1.6 Conclusions

The difference in underlying hardware and the lack of a set of suitable benchmarks makes it difficult to compare the results of the schemes described above. Furthermore, the metric most commonly used by the early researchers was tuples per second which does not mean much without the code.

Based on a detailed analysis of a set of six production system programs of various sizes, Anoop Gupta[38] concluded that the potential parallelism in the match phase of production system programs is bounded by a factor between 20 and 30. He further concluded that fine-grain decomposition is required to achieve significant speedup and that the communication and scheduling overheads of such a decomposition limit the achievable speedups to less than a factor of about 20. Keeping in mind the features of contemporary production system languages and the nature of the tasks for which he expected them to be used, he concluded that this

result is widely applicable and not just limited to the test programs used in the analysis. The primary cause of this program-independent bound is the uniformly high frequency of barrier synchronizations in parallel execution of the match phase. Since there is little work to be done in each match phase, only a small number of processors can be gainfully employed.

9.2 Parallel firings

Several research efforts have attempted to devise techniques to determine the set of instantiations that can be fired in parallel without violating the semantics of the program. All of these are variations on the analysis proposed by Ishida and Stolfo in their seminal paper[56]. They proposed a compile-time analysis, based on a dependency-graph, to determine which cycles could be safely combined. The nodes in their dependency-graph correspond to individual productions and the links represent the dependency information available at compile-time. There is a link between two nodes, A and B, if there is a dependency between *any* pair of instantiations of the corresponding productions. If the actions of production A modify a tuple which of the same type as one of the tuples matched by production B, the graph has a directed *read-write* link between A and B. If the actions of two productions modify a tuple of the same type, the graph has a bidirectional *write-write* link between the two. Figure 9.2 shows the productions for a xor-gate simulator and the dependency-graph for it. In addition to the direct dependencies represented by the links, two productions can have an indirect dependency between them if one of them lies in the transitive closure (over the dependency links) of the other. Two productions are independent if neither lies in the transitive closure of the other. This analysis uses only the type of the tuples. Subsequent efforts by Tenorio and Moldovan[119], Miranker *et al.* [81] and Schmolze and Goel[101] have refined the analysis by taking advantage of the constant literals that occur in both the if-part and the then-part of the productions.

Compile-time dependency analysis of production system programs is seriously limited by the data-dependent nature of the computation. Since the compiler has no information about the run-time contents of the tuple-space, it is forced to be overly conservative and often fails to prove the independence of productions that are obviously independent. For example, consider the productions of the xor-gate simulator shown in Figure 9.2. Simulation of every xor-gate can be done in parallel. So, we would expect that all instantiations of xor-gate-on and xor-gate-off can be executed in parallel. However, a conservative analysis is unable to detect this as:

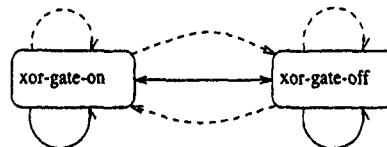
- both productions modify tuples of type line, therefore there is a write-write dependency between them and
- since both productions also match tuples of type line, there is a bidirectional read-write dependency between them.

```

(p xor-gate-on
  (xor-gate ^in1 <input1> ^in2 <input2> ^out <output>)
  (line ^id <input1> ^value <>)
  (line ^id <input2> ^value <>)
  (line ^id <output>)
  -->
  (modify 4 ^value 0))

(p xor-gate-off
  (xor-gate ^in1 <input1> ^in2 <input2> ^out <output>)
  (line ^id <input1> ^value <>)
  (line ^id <input2> ^value <>)
  (line ^id <output>)
  -->
  (modify 4 ^value 1))

```



Dashed lines represent read-write dependencies and solid lines represent write-write dependencies

Figure 9.2: Dependency-graph for xor-gate simulator

This is not a limitation of this particular technique for compile-time analysis, rather of compile-time analysis itself. To be able to determine that `xor-gate-on` and `xor-gate-off` are independent, the compiler needs to prove that is that each tuple of type `line` occurs on the output of one and only one `xor-gate`. In the absence of information about the run-time contents of the tuple-space, there is no way for a compiler to prove this.

Therefore it is not surprising that compile-time dependency analysis has had limited success. Several publications have claimed small reductions in the number of *msa* cycles [33, 56, 81, 101, 107, 119]. The stated assumption of these publications is that given enough processors,

the time taken for each cycle would be the same. The unstated assumption is that all tasks that are generated from the firing of different instantiations can be performed independently. These assumptions are unsound as shown by the results presented in [2, 38, 42, 113]. These results indicate that inter-task dependencies are a major limitation on speedups in production system programs. As the results for *waltz* show (see Chapter 5), these assumptions do not hold even for simple programs written with commonly used programming idioms. For the largest data set used in this investigation, 59524 instantiations were fired in 32 cycles yielding a 1860 fold reduction in the number of cycles. However, a detailed simulation indicates a speedup of only 17.9 fold using 100 processors. The results also indicate that the speedups will not increase significantly for larger configurations. Therefore, a measure based on the number of cycles is inherently flawed.

To work around the limitations of compile-time analysis, Oshisanwo and Dasiewicz suggested a run-time analysis of instantiations[91]. Their scheme inserts a dependency analysis phase between the match and select phases. This analysis uses the selection procedure of the sequential language to impose a total order on the instantiations currently in the conflict set. It then checks every instantiation for interference with instantiations that are above it in the ordering. If an instantiation does not interfere with any of the preceding instantiations, it is fired. A similar scheme has been proposed by Ishida[55].

Since the only information available to such schemes is the set of instantiations that are present in the conflict set during one cycle, they are able to detect only direct dependencies. This means they are able to identify the instantiations which modify tuples used to generate other instantiations in the conflict set. In the absence of information about future instantiations, they are unable to detect indirect dependencies. For example, suppose the conflict set in the first cycle has two non-interfering instantiations, A and B. A parallelizing implementation based on run-time analyses like those mentioned above would fire both A and B. Now suppose the firing of A leads to the generation of C which interferes with B, say, it deletes a tuple contained in B. Under sequential semantics, it is possible that A is selected for firing in the first cycle and C in the second. In that case, B is deleted from the conflict set without being fired. It is possible to construct such examples for any non-trivial selection procedure (Figure 9.3 shows an example for the OPSS selection procedure). Therefore, without some sort of lookahead, it is not possible to guarantee that the sequential and the parallelized versions of the program generate the same result. Adding lookahead to such schemes would, in general, require the analysis procedure to explore the space of possible execution paths. Since there is no limit on the depth of the lookahead that might be needed, such analyses could be prohibitively expensive even for the small conflict sets seen in [38].

Compile-time as well run-time dependency analysis was used by Kuo *et. al* in the CREL implementation on a Sequent Symmetry [64]. In addition to performing dependency analysis to determine which instantiations can be fired in parallel, they modified the semantics of the OPSS language to eliminate the sequentializing recency feature from the selection procedure.

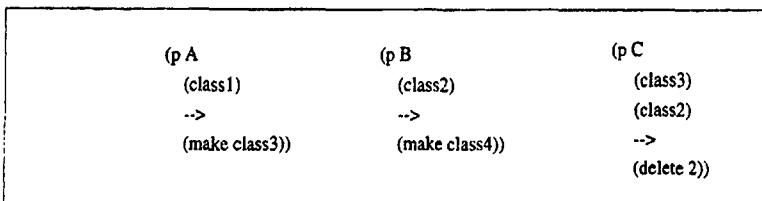


Figure 9.3: OPSS example of indirect dependency

They report speedups between two and six fold using 15 processors. They conclude that the primary reason for the low speedup was the programming style which limited the ability of the implementation to prove independence between productions. The benchmarks used in their study include *life* and *waltz* whose optimized versions have been used in the PPL experiments reported in Chapters 4 and 5. The results from these experiments indicate speedups up to 19 fold for *waltz* and up to 30 fold for *life* over the optimized uniprocessor versions. As mentioned in Sections 4.1.2 and 4.1.3, the optimized version of *life* was between 17 and 18 times faster than the original version and the optimized version of *waltz* was up to 1.9 times faster than the original version.

A commonly used idiom in production system programs is the use of a *context* tuple to direct the control-flow. Productions in programs that make use of this idiom test the context element; only the productions that test for the context currently in the tuple-space can possibly match. Kuo *et. al*[65] proposed a variation of the Ishida-Stolfo analysis where the entities being scheduled were *contexts* and not individual productions. The contexts that were proved to be independent by this analysis were executed in parallel. The primary limitation of this scheme is the fact that the programs it analyzes are written in a sequential language. The context-tuple idiom is used most often to enforce a particular execution path. This often results in spurious inter-context dependencies.

Harvey *et. al* hand-parallelized the execution of the first two phases of SPAM[75], a knowledge-based image recognition system. They decomposed the computation into large tasks that could be run independently of each other and used a task queue to schedule them on an Encore Multimax[48]. They reported speedups up to 12.5 fold using 14 processors on a single Multimax and up to 15 fold using 23 processors on a pair of Multimaxes which shared virtual memory using a *netmemory server* [31]. From an analysis of the tasks and their dependencies, they concluded that it was possible to achieve between 50 and 100 fold speedup if enough processors were available. The *spam* benchmark used in the PPL experiments corresponds to the first *three* phases of SPAM. As discussed in Section 4.1.5, the first two phases of SPAM consist of fully parallel triply nested loops whereas the third phase contains two non-parallelizable

loops. In spite of this, *spam* achieves speedups up to 52 fold using 100 processors. This is explained by the fact that the PPL version of *spam* parallelizes all the loops in the first two phases whereas the decomposition used by Harvey *et. al* corresponds to parallelizing only the outer two loops in both phases.

9.3 Parallel Production Languages

Parallel production languages can be classified into two groups - synchronous and asynchronous. Synchronous languages enforce a strict match-select-act cycle, instantiations being fired only during the act phase. Asynchronous languages allow instantiations to be fired at any time.

Given the absence of explicit control structures in the production system paradigm, it is much easier to program synchronous languages than asynchronous languages. Asynchronous instantiation firings can lead to non-deterministic behavior and race conditions for non-monotonic tuple-spaces. As a result, most production system languages are synchronous.

Dan Neiman proposed an asynchronous parallel production language[86]. To help achieve deterministic behavior, his language provides multiple-reader-single-writer locks on individual tuples. When an instantiation is generated, it attempts to acquire read locks on all tuples contained in it and write locks for tuples it intends to modify or delete. If it is unable to acquire these locks, the instantiation waits on these locks. The locks are released after the instantiation has been fired. This scheme does not guarantee deterministic behavior. For example, consider the productions and the tuple-space in Figure 9.4. Two instantiations of *get-new-order* are generated - one that selects the order to launch missiles on Timbuctoo and the other that cancels all pending missile launch orders. If the first instantiation is generated earlier and fired, it causes an instantiation of *take-action-1* to be fired. On the other hand, if the second instantiation is generated earlier, it causes an instantiation of *take-action-2* to be fired. Therefore, the result of the computation, which determines whether Timbuctoo survives, depends on the relative speeds of the processors.

Synchronous production system languages can be subclassified into two groups: collection-oriented (or set-oriented) languages and tuple-oriented languages. Tuple-oriented languages dictate that each instantiation of a production must contain exactly one matching tuple for every non-negated condition in the production and no matching tuples for every negated condition in the production. A variable in a production is, therefore, bound to a single value and the actions in the *then-part* of a production operate on individual tuples. Collection-oriented languages allow an instantiation to contain a collection of tuples corresponding to every non-negated condition. Instead of containing *one* sequence of tuples that jointly satisfy the conjunction of conditions in the *if-part*, such instantiations would contain *all* sequences of tuples that jointly satisfy the conditions. For example, consider the production and the tuple-space in Figure 9.5. In this example, there are three data items whose value is below the threshold (16 and 40 and

```
(p get-new-order
  (order ^id <order> ^status pending)
  (current-order ^id {<current> <> <order>})
  (order ^id <current> ^status completed)
  -->
  (modify 2 ^id <order>))

(p take-action-1
  (current-order ^id <order>)
  (order ^id <order> ^action launch-missiles ^target <> ^status pending)
  -->
  (modify 2 ^status completed)
  (call (launch_missiles <>)))

(p take-action-2
  (current-order ^id <order>)
  (order ^id <order> ^action avoid-launching-missiles ^status pending)
  (order ^id ^action launch-missiles)
  -->
  (remove 3))

t1: (order ^id 1 ^status completed ^action verify-readiness)
t2: (order ^id 2 ^status pending ^action launch-missiles ^target Timbuctoo)
t3: (order ^id 3 ^status pending ^action avoid-launching-missiles)
t4: (current-order ^id 1)
```

Figure 9.4: Example of nondeterminism in asynchronous firings.

```

(p apply-threshold
  (threshold ^value <v>)
  (data-item ^value (<data> <<v>>)))
->
(write "Accepted value " <data> "for threshold " <v>))

t1: (threshold ^value 80)
t2: (data-item ^value 16)
t3: (data-item ^value 40)
t4: (data-item ^value 72)

```

Figure 9.5: Example to illustrate differences between synchronous languages

72). In a tuple-oriented parallel language, separate instantiations are generated for each data-item below the threshold, all of which are fired in parallel. In a collection-oriented language, all the instantiations for a single threshold are grouped together. Therefore, only one instantiation is generated for the tuple-space in Figure 9.5.

There is only one sequencing point in the execution of a synchronous production system language – the select phase, which selects the operation(s) to be performed in each cycle. It consists of two parts. The first part uses an ordering relation between instantiations to order the conflict set and the second part determines and extracts its *dominant subset*.¹ Instantiations in the dominant subset are fired in the act phase. The features that distinguish different parallel production languages are the kind of ordering relations permitted and the manner in which they are to be specified.

Ordering relations can be trivial or non-trivial. A trivial ordering relation would either totally order all the instantiations, that is, the number of levels in the order equals the number of instantiations, or order none of the instantiations, that is, all instantiations are considered to be at the same level. The former corresponds to sequential execution and is unsuitable for a parallel language. The latter would fire all instantiations generated and has been used in Soar[66] which uses productions to model associative memory. One of the main reasons to select a subset of the generated instantiations is to ensure that conflicting tuples are not added to the tuple-space. To deal with this problem, Soar uses a separate decision procedure that is run after all matching instantiations have been fired. Among other things, the decision procedure identifies and deals with conflicting tuples.

¹The dominant subset of D_S of a set S ordered by the relation \preceq is defined as $D_S = \{t | \forall y \in S \ y \neq t \wedge y \not\preceq t\}$

There are two ways in which a non-trivial ordering relation can be specified: declarative, that is, by explicit enumeration and procedural, that is by a procedure which given two instantiations either indicates the order between them or indicates that they are incomparable.

Explicit enumeration: The set of all instantiations is infinite. Therefore, it is impossible to explicitly enumerate the relationship between all pairs of instantiations. However, the set of productions in a program is finite. It is possible to explicitly enumerate a partial order relation from the set of productions in the program to itself and this order can be extended to the instantiations of these productions. A priority is associated with each production and gets propagated to all its instantiations. Other possibilities include grouping productions and assigning priorities to groups, an explicit ordering sublanguage and so forth. Priority-based schemes have been used in Ariel[44], and the Starburst rule language[126]. Given its static nature, this scheme is able to specify the relationship between instantiations of different productions. It is unable to specify an order on instantiations of the same production and they must be assumed to be incomparable. In other words, they must be assumed to not interfere with themselves (for example of a simple production that interferes with itself, see Figure 9.6). Since it is rarely the case that all productions of a program are self-independent, this inflexibility renders explicit enumeration undesirable for specifying ordering relations between instantiations. However, if the production system program is partitioned into production-sets, explicit enumeration can be used to specify the dependencies between different production-sets. Production-sets that have no mutual dependencies can be executed independently. This has been used in the OPRL language[128]. An interesting approach to explicit enumeration has been taken by de Maindreville *et. al* in the RDL/C system[15]. RDL/C provides a regular expression-like control language to specify ordering between productions. As shown by the programs written in PPL (see Chapters 4 and 5 and Appendix E), parallel execution of subprograms can be achieved within the production system language itself without the baggage of an additional control language.

Procedural ordering: Procedural ordering delays ordering decisions until run-time when the instantiations are available. It is more powerful than explicit enumeration since it is able to order instantiations of the same productions. It can use a wide variety of properties of instantiations, ranging from their size and the timetags of the constituent tuples to values in constituent tuples.

Procedural ordering can be supported in two ways – by providing a fixed ordering procedure as a part of the language or by providing a sublanguage to specify program-specific ordering procedures. Since a fixed ordering procedure has no knowledge about the productions, it can depend only on structural properties, like the size of instantiations, and on universal attributes like the timetags of the tuples. Therefore, it can be expected to order instantiations quickly, for reasonable ordering relations. Program-specific ordering procedures can be more discriminating than a fixed procedure by taking advantage of program-specific information. However, program-specific ordering procedures can be arbitrarily complex. This can significantly increase the time needed to order the instantiations as well as make such programs much more

```

(p paint-red-ball
  (ball ^color blue)
  (ball ^color red)
  ->
  (modify 2 ^color blue))

T1: (ball ^color blue)
T2: (ball ^color blue)
T3: (ball ^color red)

Instantiations: <T1,T3>, <T2,T3>

```

Figure 9.6: Simple production that interferes with itself

difficult to comprehend. In this author's opinion, the flexibility provided by program-specific procedures is not sufficient to offset its disadvantages. The PARULEL language [107] provides a production system sublanguage to specify the ordering procedure. In this author's opinion, most programs that have been written using PARULEL can be easily and more concisely written in a language based on a fixed ordering procedure. Hernandez and Stolfo present two PARULEL programs in [50]. Both of these programs have been easily parallelized in PPL which has a fixed ordering procedure. PPL code for these programs can be found in Appendix E.

The discussion above applies to both tuple-oriented and collection-oriented languages. However, the extent to which individual languages support collections provides additional discrimination between collection-oriented languages. The ideal collection-oriented language would allow collections to be used wherever individual tuples or values can be used. However, many collection-oriented production languages allow collections to be used only in certain contexts which are explicitly flagged by use of special syntax.

Several collection-oriented production system languages allow only selected conditions to match collections of tuples. Conditions that can match collections of tuples are flagged by use of special keywords, for example the FORALL keyword used in *Herbal*[122], and bracketing constructs, for example, the square brackets used in extended C5[36] and OPRL[128]. On the other hand, database rule languages, all of which have been derived from the SQL query language[53], allow every condition to match a collection of tuples. Examples include RPL[21], the Starburst Rule Language[126], RDL/C[15]. The same distinction exists on the action side. Collection-oriented languages derived from OPS5-like languages allow only special system-

defined operations (like `set-modify`, `set-remove` and `foreach`) to process collections, while database rule languages allow collections to be processed by any available procedure. COPL is the only collection-oriented production system language that allows collections to be used anywhere a scalar can. Other distinguishing features of COPL include easy interfacing to routines in procedural languages and the `update` operation that zips together a collection of tuples and a collection of values.

9.4 Reducing Combinatorics

9.4.1 Relaxing the completeness constraint

A match algorithm can relax the completeness constraint by generating only a subset of the instantiations matched. From a programming point of view, it is reasonable to not generate an instantiation if and only if it can be proved that it will not belong to the dominant set. In other words, an instantiation need not be generated if and only if it can be shown that there exists at least one other instantiation that dominates it. Similarly, a partial match need not be extended if and only if it can be shown that there exists another partial match such that all instantiations generated by extending the former are dominated by all instantiations generated by extending the latter. This requires that the ordering procedure be integrated into the match algorithm. Tzvieli *et. al*[121] and Miranker *et. al*[79] independently proposed integrating the OPS5 selection strategy into the match algorithm. The latter proposal was later extended into the LEAPS match algorithm and was incorporated into the `ops5c` compiler[80].

If only one instantiation is fired at a time, generating only the dominant instantiation(s) and the partial matches that lead to them can limit the growth in the number of instantiations and partial matches. This technique is likely to be particularly effective in programming idioms that generate a host of instantiations and fire only small subset of them. Examples include using the conflict set to select an item from a collection and accumulation loops (see Section 5.2.4 for a discussion on accumulation loops). As discussed in Section 8.5.7, these idioms are poor programming practice and can be eliminated in collection-oriented languages. Since this approach does not relax the correctness constraint, it is possible that all matched instantiations are fired and will be generated. For example, consider the production in Figure 9.7. This production has been taken from the `make-teams` program described in Section 8.1.1. Since, all of the remaining constraints remain in effect, this approach does not improve the worst-case complexity of the match procedure.

```

(parl make-team
  (goal ^name create-teams)
  (person ^id <id1> ^expertise hardware ^previous-project <project> ^score <v1>)
  (person ^id <id2> ^expertise operating-system ^score <v2>)
  (person ^id <id3> ^expertise networks ^score <v3>)
  (person ^id <id4> ^expertise compilers ^previous-project <project> ^score <v4>)
  -->
  (make team ^hardware <id1> ^operating-systems <id2> ^networks <id3>
        ^compilers <id4> ^score (compute <v1> + <v2> + <v3> + <v4>))

(parl create-teams
  (goal ^name select-team)
  (team ^score > 8 ^select-status nil)
  -->
  (modify 2 ^select-status selected))

(parl count-teams
  (goal ^name count-teams)
  (team ^select-status selected)
  (count ^value <value>)
  -->
  (modify 2 ^select-status counted)
  (modify 3 ^value (compute <value> + 1)))

```

Figure 9.7: PPL productions for make-teams

9.4.2 Reduce the power of each condition

If the number of tuples that match individual conditions can be bound by a small constant, the number of instantiations and partial matches can no longer increase combinatorially with the growth in the size of the tuple-space. The *unique-attributes* scheme proposed by Milind Tambe in his thesis[112] limits the number of tuples that can match individual conditions to one. For example, consider the production and the tuple-space in Figure 9.8. This production attempts to find all points that are three links away from the current position. The graph represented by the tuple-space is also shown in Figure 9.8. For this graph, the second and fourth conditions

are matched by two tuples each, (t_2, t_3) and (t_7, t_8) respectively. The number of tuples matching each condition can be limited to one if the connected-to field is split in to several fields such that each point has only one value for each field. Since the graph has only rectilinear links, the connected field can be split into the up, down, left and right fields. Figure 9.9 shows what the representation of the graph would be in this scheme. Since each point has only one link in any of these directions, the number of tuples matching suitably modified conditions is at most one. However, note that it is no longer possible to encode the task of finding all paths of length three from the current position in a single production; a separate production is required for every path of length three. Figure 9.9 shows one such production.

The unique-attribute approach limits the number of instantiations and partial matches for each production to the number of conditions it has. Therefore, it is able to guarantee that the cost of matching a production is linear in the number of conditions. However, the restriction of a single instantiation per production may require an exponential number of productions to perform the same task. In the above example, if the task is to find *all* paths of length 3 from the current position, four productions and their corresponding instantiations will be necessary.

The unique-attribute approach restricts both the tests that can appear in conditions and the values that can appear in the tuples to limit the number of tuples that can match individual conditions. By doing so, it is able to guarantee that no more than a linear number of instantiations and partial matches are generated per production. Other, less comprehensive, schemes have attempted to achieve the same end by restricting only one of the two, either the tests that appear in the conditions or the values that appear in the tuples. As a result, these schemes do not reduce the number of instantiations and partial matches generated. However, they are able to reduce the total number of comparisons performed during the match procedure.

copy-and-constrain: Copy-and-constrain[92] attempts to limit the number of tuples matching individual conditions by restricting the tests that can appear in the conditions. It places no restrictions on the values that can appear in the tuples. Consider the production and the tuple-space in Figure 9.10. In this case, each condition individually matches all five tuples. The total number of comparisons needed is $4 \times 4 + 4 \times 3 = 28$.² The number of instantiations generated is four, two each for red and blue pieces. Copy-and-constrain takes advantage of knowledge of the values that will appear in the tuples to replace the inter-condition test that matches the colors of the pieces (the shared variable $\langle x \rangle$) by consistent intra-condition tests that check for a specific color. The converted productions are shown in Figure 9.11. In this case, only two tuples match each condition. The total number of comparisons is reduced to $4 + 4 \times 2 = 12$ ³ but the number of instantiations is still four – two for the red pieces and two for the blue pieces. In general, it is not necessary to know the set of values that appear in the tuples. It is possible to

²Each tuple is compared with all four tuples for the test $\langle id2 \rangle \leftrightarrow \langle id1 \rangle$ and with the other three tuples for matching the color.

³Each tuple is tested for its color and each tuple is compared against itself and the other tuple of the same color.

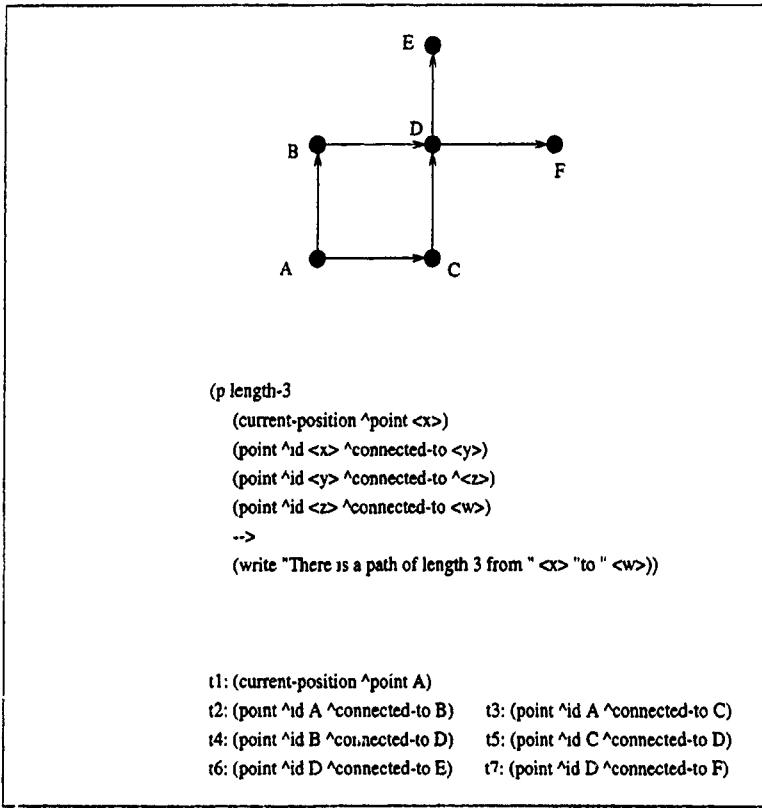


Figure 9.8: Example production and tuple-space for multi-attribute representation

```

(p length-up-right-right
  (current-position ^point <x>)
  (point ^id <x> ^up <y>)
  (point ^id <y> ^right <z>)
  (point ^id <z> ^right <w>)
  -->
  (write "There is a path of length 3 from " <x> "to " <w>))

t1: (current-position ^point A)
t2: (point ^id A ^up B)          t3: (point ^id A ^right C)
t4: (point ^id B ^right D)      t5: (point ^id C ^up D)
t6: (point ^id D ^up E)          t7: (point ^id D ^right F)

```

Figure 9.9: Tuple-space and one of the productions for the unique-attribute representation

hash the timetags of the tuples, which form a universal key for the tuples, into a small number of buckets and generate a separate production corresponding to every bucket. Since multiple versions of a production are generated for every such conversion, the number of productions needed to perform a task is $O(b^c)$ where b is an upper bound on the number of buckets and c is the number of times the copy-and-constrain conversion is applied. Note that in spite of this blowup in the number of productions, copy-and-constrain is unable to guarantee that the number of instantiations and partial matches is subexponential in the number of conditions.

Data partitioning: Data partitioning attempts to limit the number of tuples that match individual conditions *at any given time* by partitioning the tuple-space. It places no restrictions on the tests that may appear in the conditions. Consider the production and the tuple-space in Figure 9.12. The number of instantiations generated is $2 \times 4! = 48$.⁴ However, it is possible to partition the tuple-space into two partitions such that the first partition contains t_1 , t_2 and t_3 and the second partition contains the rest of the tuples. In this case, only two instantiations are generated per partition for a total of four instantiations. In general, data partitioning can reduce the number of instantiations from $O(w^n)$ to $O((w/p)^n)$ where w is the number of tuples, n is the number of conditions and p is the number of partitions. Such a reduction happens only for tasks in which tuples are grouped at most once. In the above example, firing each instantiation deletes the pool tuples contained in it. However, if the task involves combining

⁴There are two tuples of type `need` which match the first condition; the other two conditions can be matched by any permutation of the four pool tuples.

```
(p connect-pieces
  (piece ^id <id1> ^color <x>)
  (piece ^id {id2} <id1> ^color <x>)
  ->
  (make pair ^first <id1> ^second <id2>))

t1: (piece ^id 1 ^color red)
t2: (piece ^id 2 ^color red)
t3: (piece ^id 3 ^color blue)
t4: (piece ^id 4 ^color blue)
```

(adapted from Figure 1 in [92])

Figure 9.10: Before the application of copy-and-constrain

```
(p connect-red-pieces
  (piece ^id <id1> ^color red)
  (piece ^id {<id2> <id1>} ^color red)
  ->
  (make pair ^first <id1> ^second <id2>))

(p connect-blue-pieces
  (piece ^id <id1> ^color blue)
  (piece ^id {<id2> <id1>} ^color blue)
  ->
  (make pair ^first <id1> ^second <id2>))
```

Figure 9.11: After the application of copy-and-constrain

or comparing each tuple with every other tuple in the data-set, there is no reduction in the number of instantiations and partial matches generated. For example, consider the production and the tuple-space in Figure 9.13. In this case, every `taxiway` tuple has to be compared with every `runway` tuple. Therefore, partitioning the data set does not reduce the total number of instantiations and partial matches. Even though there may be no reduction in the total number of instantiations and partial matches generated *at any given time*. This reduces the stress on the state-maintenance algorithms. Therefore, it might be preferable to partition the data even if it does not reduce the number of instantiations.

```
(p two-rule-combo
  (need ^contract-id <cid> ^total <>)
  (pool ^id <pid1> ^value <p1>)
  (pool ^id (<pid2> <> <pid1>) ^value <p2>)
  ->
  (remove 2 3)
  (modify 1 ^total (compute <> - (<p1> + <p2>)))))

t1: (need ^contract-id 1 ^total 1000)
t2: (pool ^id 101 ^value 200)
t3: (pool ^id 102 ^value 300)
t4: (need ^contract-id 2 ^total 1000)
t5: (pool ^id 201 ^value 200)
t6: (pool ^id 202 ^value 300)
```

(adapted from Figure 1 in [107])

Figure 9.12: Data partitioning example

An important problem with data partitioning is that in many cases, it may reduce the quality of the solution. In the example shown in Figure 9.12, processing the data as a whole completes allocates one of the contracts, that is, the `total` for one of the `need` tuples goes to zero. Processing the data on a per-partition basis leaves both contracts partially allocated which may not be desirable.

```

(p LCC--runways-are-orthogonal-to-taxiways
  (lcc-stage ^name apply-constraint)
  (lcc-rule-constants ^rulename runways-are-orthogonal-to-taxiways
    ^min <min> ^max <max> ^bound <bound>)
  (fragment ^hypothesis runway ^identifier <id0>)
  (fragment ^hypothesis taxiway ^identifier {<id1> <> <id0>})
  ->
  (make lcc-match-score ^rulename runways-are-orthogonal-to-taxiways
    ^result (spam_lcc_do_geometric_test 12 <min> <max> <bound> 10000)
    ^from <id0> ^to <id1>))

```

(adapted from the LCC phase of SPAM[75])

Figure 9.13: Case whether data partitioning does not help

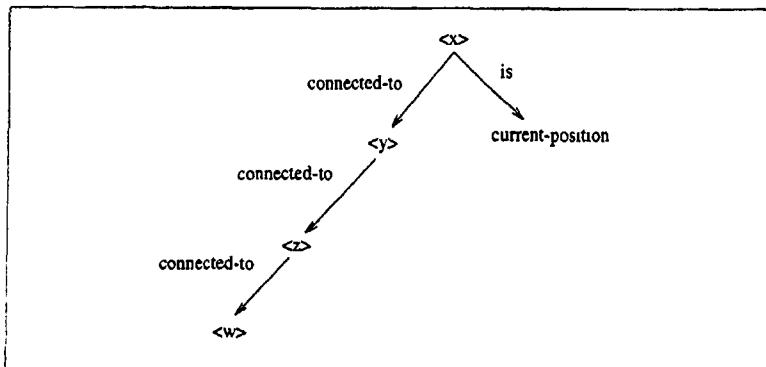
9.4.3 Relax the correctness constraint

Consider the production and the tuple-space in Figure 9.8. The problem solved by this production is to find all points that are three hops away from the current position. The write action prints out the current position and the set of the points found. In this case, the paths from the current position are not relevant. All that matters is that there exists *some* path from the current position. In terms of the production, the information required is the bindings of the variables $\langle x \rangle$ and $\langle w \rangle$. Therefore, an instantiation that binds $\langle x \rangle$ to A, $\langle y \rangle$ to the set {B,C}, $\langle z \rangle$ to D and $\langle w \rangle$ to the set {E,F} is adequate for this problem. In general, this approach, referred to as *instantiation-less match* [117], binds each variable to a set of values ignoring the relationship between individual values in the sets. By viewing tuples as constraints between variables instantiation-less match can be directly mapped to the problem of finding consistent bindings for constraint-satisfaction problems[20]. For example, the production in Figure 9.8 can be viewed as the constraint graph shown in Figure 9.14. Since instantiation-less match does not restrict the organization of the inter-condition tests, it is equivalent to the problem of finding consistent bindings for unrestricted constraint graphs[117]. Dechter and Pearl [20] show that this problem is NP-hard. However, if suitable restrictions are placed on the organization of the inter-condition tests, for example if the constraint-graph corresponding to the production is a tree, a polynomial bound can be established on the number of instantiations and partial matches[117]. In terms of production syntax, a tree organization corresponds to a production where no condition tests two previously bound variables. The production in Figure 9.8 is an example of a tree-structured organization. However, it will no longer be a tree-structured

production if the condition

(point ^id <x> ^connected-to <z>)

is appended to the conditions already present.



(taken from Figure 14 in [117])

Figure 9.14: Constraint graph for the example production

The collection-oriented match approach presented in this thesis is a derivative of instantiation-less match. Depending on the tests occurring in the conditions and the values occurring in the tuples, it is able to position itself anywhere in the spectrum between instantiation-less match and classical tuple-oriented match. On one hand, if there is no need to discriminate between different tuples that match individual conditions, it operates like instantiation-less match. On the other hand, if the tests in the conditions need to discriminate between all the tuples matching individual conditions, it degenerates into tuple-oriented match.

Chapter 10

Conclusions

This chapter presents the conclusions of this dissertation and discusses some avenues for future research.

10.1 Primary conclusions

1. In general, there is no program-independent bound on the speedup that can be achieved by parallel production system programs.
2. Speedups in parallel production systems can scale with data set size. That is, parallelism is a feasible solution for the problem of dealing with large tuple-spaces.
3. Effective parallelization of production system programs requires information about the run-time contents of the tuple-space.
4. A fixed ordering procedure combined with program annotations that specify which instantiations are comparable is sufficient for the expression of parallelism in production system programs.
5. It is sometimes possible to tame the combinatorial explosion in the number of instantiations and partial matches as the data set size grows without restricting either the expressiveness of the productions or the contents of the tuple-space. That is, collection-oriented match is a feasible solution for the problem of dealing with large tuple-spaces.
6. The rate of growth in number of instantiations and partial matches depends on the specificity of the tests in the production *relative* to the contents of the tuple-space

The following paragraphs discuss these conclusions at greater length.

No program-independent bound on speedups: Based on a detailed analysis of a large set of production system programs, Anoop Gupta[38] concluded that there is an empirical *program-independent* bound of between 20 and 30 fold on the parallelism available in production system programs. He further concluded that fine-grain decomposition is required to achieve significant speedup and that the communication and scheduling overheads of such decompositions limit the speedup to under 20 fold. Results of other investigations have been consistent with this conclusion[2, 42, 64, 65, 113]. Based on the expectation that the nature of the tasks performed by production system programs will be similar to those performed by his benchmarks, Gupta concluded that using explicitly parallel languages would not substantially increase speedups. This expectation has not been met. The applications that production systems are currently being considered for process orders of magnitude more data than the programs included in Gupta's benchmark suite.

Results presented in this dissertation indicate that there is no program-independent bound on the speedups that can be achieved by parallel production system programs. The benchmarks used in this investigation achieve up to 76 fold speedup with 100 processors and up to 115 fold speedup with 200 processors. The detailed results in Section 5.2 show that these numbers are not an upper bound on the speedup that can be achieved by parallel production system programs and that, for some programs, larger speedups can be expected if the data set size is increased further. The analysis in Section 5.1.1 identifies small task size, non-parallelizable loops and large cross-products arising due to the use of sequencing tuples as the primary limitations on speedups in parallel production system programs. These limitations can be alleviated, if not eliminated, by using collection-oriented match algorithms and collection-oriented languages (for details, see Section 5.3.6).

Speedups can scale with data set size: For speedups to grow with data set size, the available parallelism must be proportional to the data set size. Programs that process variable-sized data sets can be expected to contain loops whose number of iterations depends on the size of the data set. If these loops are parallelizable, then different iterations of these loops can be mapped to independent instantiations which can be fired in parallel. As the data set size grows, so does the number of instantiations that are fired in parallel. If the subsequent match phase does not contain large cross-products arising due to the use of sequencing tuples an increase in the average number of instantiations fired per cycle will lead to an increase in the speedup.¹

Results presented in Section 5.2 show that in many cases, the speedup achieved with a given configuration grows with the data set size, the rate of growth depending on program characteristics. The only benchmark for which this does not happen is *life*. In this case, the scalability of the speedups is limited by the nature of the data set which consists of repetitions of a basic

¹See Section 5.1.3 for a discussion of sequencing bottlenecks arising due to large cross-products and Section 5.3.5 for guidelines for eliminating such bottlenecks

pattern of thirty-two cells of which only two ever change status. The loops that process updates to these cells can be executed in parallel but the loops that print the entire pattern cannot. If the fraction of mutating cells is significantly larger and the number of generations computed are significantly greater, the fraction of time spent in the sequential print loop will not be as important as it is in this benchmark. In such cases, the speedup can be expected to scale with the data set size.

Explicit specification of parallelism necessary: The production system computational model is *synchronous*. In each match-select-act cycle, the match phase must be completed before the instantiations to be fired can be selected. Parallel implementations of production system programs need a barrier synchronization to ensure this. There are three reasons for this: implementation of negated conditions, global selection policies and the need for non-monotonic tuple-spaces (see Section 3.1.1 for details). Since the average size of match-select-act cycles in a sequential production system language is small, independent of the data set size, it is necessary to combine multiple match-select-act cycles and share a single barrier synchronization between them. To preserve correctness, a set of cycles can be automatically combined if and only if the implementation is able to prove that there are no dependencies between the instantiations being fired in these cycles (see Section 3.1.2.2 for a discussion of dependencies between instantiations). A compile-time analysis has access to only the productions. To prove that two productions are independent, it has to show that all possible instantiations of the two productions have no dependencies between them. Since this analysis has no information about the run-time contents of the tuple-space, it is forced to be overly conservative and often fails to prove the independence of productions that are obviously independent. Section 3.1.2.2 shows examples of simple and obviously independent productions whose independence cannot be proved at compile-time. A run-time analysis operates within the context of a particular tuple-space. Since it does not have to consider all possible tuple-spaces that might be presented to the program, it can be less conservative than compile-time analysis. To fire two instantiations in parallel, a run-time analysis has to prove that this does not change the final result of the program. In the worst case, this might require exploration of all possible execution paths. In the absence of such lookahead, run-time analysis is limited to detection of direct dependencies. Relying on direct dependencies alone can compromise correctness. Section 3.1.2.2 shows a simple program in which firing instantiations that have no direct dependency yields an incorrect result.

Failure of both classes of analyses can be attributed to the data-dependent nature of the computation in production system programs combined with a lack of information about the contents of the tuple-space. In the absence of user specification, there is no way for a production system implementation to gain this information. Section 3.2 discusses several ways in which this information can be made available to the implementation.

Simple constructs are expressive enough: The primary decision to be made in the design of a parallel production system language is the manner in which information about the contents

of the tuple-space is to be provided. This dissertation proposes a fixed ordering procedure for instantiations combined with program annotations to indicate which instantiations are comparable. A fixed ordering procedure has no knowledge about the productions, it can depend only on structural properties, like the size of instantiations, and on universal attributes like the timetags of the tuples. However, since the annotations are specified at compile-time, they cannot discriminate between different instantiations of a single production. Either all instantiations of a production are incomparable or none of them are. Program-specific ordering procedures that are executed at run-time have access to the instantiations and can be more discriminating. However, program-specific ordering procedures can be arbitrarily complex. This can significantly increase the time needed to order the instantiations as well as make such programs much more difficult to comprehend. The results presented in Chapter 5 indicate that the simple constructs proposed in this dissertation are sufficient to express the parallelism available in a variety of production system programs and the additional flexibility provided by program-specific ordering procedures may not be necessary. Hernandez and Stolfo present two programs, which use program-specific ordering procedures, in [50]. Appendix E contains PPL versions of both these programs.

Combinatorial explosion can be tamed without giving up expressiveness: Researchers seeking to eliminate the combinatorial explosion in the number of instantiations and partial matches that occurs with a growth in the tuple-space size, have proposed restricting both the tests that can appear in productions and the values that can appear in the tuple-space [112, 117]. The unique-attribute approach limits the number of tuples that can match each condition to one. By doing so, it is able to guarantee that at most one instantiation and at most n partial matches, where n is the number of conditions, are generated per production. The instantiation-less match proposal takes a similar approach but places weaker restrictions on the productions and the tuples. The collection-oriented match approach proposed in this dissertation does not place any *a priori* restrictions on either the productions or the tuples. Depending on the tests occurring in the conditions and the values occurring in the tuples, it is able to position itself anywhere in the spectrum between instantiation-less match and classical tuple-oriented match. If the tests in the productions do not discriminate between tuples, it operates like instantiation-less match; if the tests in the productions need full discrimination between tuples that match individual conditions, it degenerates into tuple-oriented match. The results presented in Chapter 8 provide an illustration.

Rate of growth in the number of instantiations: For a collection-oriented match algorithm, the number of instantiations of a production depends on the extent to which the tuples matching individual conditions can be grouped together. If all the tuples matching every condition form a single group, only one instantiation is generated for every production. On the other hand, if every tuple forms its own group, a collection-oriented algorithm reduces to its tuple-oriented analogue. In other words, the number of instantiations for a production depends on how the collections of tuples corresponding to each condition are partitioned, or *fragmented* (see

Section 8.4 for an example). Therefore, the rate at which the number of instantiations and tokens grows is governed by the rate at which new partitions are generated. In the ideal case, addition of tuples does not increase the fragmentation and the number of instantiations remains constant. This happens, for example, in some productions in the benchmark programs used in this investigation. In the worst case, every new tuple added increases the fragmentation. As shown in Section 8.4, the rate of growth in the number of instantiations is independent of the size of tuple-space as well as the *absolute* specificity, absolute specificity being defined as the number of partitions that would be generated for an infinite tuple-space with a uniform distribution of values. Instead, the rate is dependent on the *relative* specificity, that is, the number of partitions that would be generated for a given tuple-space.

10.2 Some general conclusions

Performance on real machines: The simulation results presented in this chapter assume a uniform memory access model. They provide a measure of the amount of parallelism available that takes the scheduling costs into consideration but does not take the details of the memory system into consideration. In effect, the speedups reported by the simulator constitute approximate upper bounds on the speedups that can be achieved on a real multiprocessor. These results help establish that there is no general program-independent limit on the speedups that can be achieved by parallel production system programs. They also help establish the feasibility of parallelism as a potential solution for the problem of dealing with large tuple-spaces. However, it is important to understand that speedups on real machines will depend on the characteristics of the machine. It is expected that for machines with a significantly different model, for example distributed memory machines, a different decomposition will be necessary to achieve good performance. Similarly, for machines with limited resources, for example, small cardinality multiprocessors, coarser decompositions will be needed so that the processors are not swamped by a large number of fine-grained tasks.

Production system languages should be collection-oriented: The results presented in this dissertation suggest that production system languages, sequential as well as parallel, should be collection-oriented. Only one instantiation can be fired at a time in sequential production system languages. Since firing an instantiation can modify only the tuples contained in the instantiation, it is not possible to atomically update large unbounded data structures, like the grid of cells for *life* or the set of lines for *circuit*, in a single cycle. Therefore, such updates have to be performed as loops spread over multiple cycles. To ensure atomicity, it is necessary to create, update and delete copies of the tuples that constitute the data structure. Collection-oriented production system languages make it possible to update an unbounded number of tuples in a single cycle. Therefore, it is possible to atomically update unbounded aggregate data structures in a single cycle. This eliminates the need for flag fields and tuple

modifications caused by resetting of these fields

The analysis presented in Chapter 5 identified three major limitations on the scalability of parallelism in parallel production system programs: small average task size, non-parallelizable loops and large cross-products arising due to the use of sequencing tuples. Each of these is closely related to, if not directly caused by, tuple-oriented semantics of production system languages (see Section 5.3.6 for details). Since a collection-oriented token contains a collection of tuples for every condition, match tasks in collection-oriented match algorithms consist of comparing collections of unbounded cardinality. This is likely to increase the average task size. A collection-oriented production system language allows the programmer to generate and manipulate collections of tuples and values. In particular, it allows her to move aggregate operations which cannot be parallelized within the production system paradigm to other languages where they can. Examples of such operations include sorting, accumulations and permutations. Finally, collection-oriented match algorithms can eliminate the sequentialization caused by large cross-products. Since there is no test between the condition that tests for the sequencing tuple and its successor, collection-oriented match algorithms will create only a single successor token containing the implicit cross-product.

Recency is unsuitable for parallel production system languages: The algorithms used in the select phase of many production system languages[11, 30, 26] use the age of tuples, as indicated by their timetags, to order the instantiations. Instantiations containing recently created tuples are placed above instantiations containing older tuples. This results in tuples being processed in a *last-in-first-out* fashion. As long as only one instantiation is fired per *msa* cycle and the actions in the firing are executed sequentially, the process of timetag generation is deterministic. In such cases, recency is a stable ordering criterion since the assignment of timetags is deterministic. Parallel production system languages permit multiple instantiations to be fired in parallel. The assignment of timetags to the tuples created by parallel firings is non-deterministic and depends on the number of processors available and their relative speeds.

The use of recency as an ordering criterion in the selection algorithm for a parallel production system language introduces three problems. First, it makes it impossible to compute the control-flow graph (or even a usable approximation) at compile time since the order in which instantiations are fired can depend on run-time data values as well as the relative speeds of the processors. Second, it is likely to lead to contention for the timetag counter causing potential serialization of tuple-space updates and thereby of the match process. Third, it makes program execution sensitive to relative processor speeds which introduces subtle race conditions Section 5.3.3 contains an example.

The rationale for the recency criterion is historical. For a long time, production system programs were used exclusively in programs modelling human cognition and reasoning. In such programs, the recency criterion is necessary to ensure responsiveness to changes in the environment[73]. In some cases, it is possible to achieve responsiveness in the absence of recency by using additional conditions. In general, responsiveness can be ensured by adding

an explicit timetag field and testing for it in the productions. This scheme has the advantage of allowing the programmer to create and use timetags as and when she needs responsiveness and does not force her to pay the cost of recency for programs or sections of programs that do not require responsiveness.

Ratio of *msa* cycles \neq speedup: Several papers describing efforts to combine multiple *msa* cycles use the number of *msa* cycles as a measure of the execution time and the ratio of the number of instantiations fired and the number of *msa* cycles as a measure of the speedup [33, 56, 81, 101, 107, 119]. The stated assumption for this is that given enough processors, the time taken for each cycle would be the same. The unstated assumption is that all tasks that are generated from the firing of different instantiations can be performed independently. These assumptions are unsound as shown by the results presented in [2, 38, 42, 113]. These results indicate that inter-task dependencies are a major limitation on speedups in production system programs. As the results for *waltz* show (see Chapter 5), these assumptions do not hold even for simple programs written with commonly used programming idioms. For the largest data set used in this investigation, 59524 instantiations were fired in 32 cycles yielding a 1860 fold reduction in the number of cycles. However, a detailed simulation indicates a speedup of only 17.9 fold using 100 processors. The results also indicate that the speedups will not increase significantly for larger configurations. Table 10.1 compares the ratio of *msa* cycles to the speedup computed by detail simulations. It shows that the ratio of *msa* cycles is neither an underestimate nor an overestimate of the speedup. Table 10.2 compares the average size of cycles for sequential and parallel versions of the benchmarks.

Program	circuit	life	waltz	hotel	spam
Ratio of <i>msa</i> cycles	12.5	196.4	1860	24.2	24.3
Speedup	29.6	23.6	17.9	115.3	35.3

Numbers presented in this table correspond to the largest data set for each benchmark.

Table 10.1: Comparison of ratio of *msa* cycles and speedups

Program	circuit	life	waltz	hotel	spam
Average cycle size (sequential)	17482	10630	5653	3580534	229870
Average cycle size (parallel)	508884	2589446	13335031	80608075	6736894
Ratio	29.1	243.6	224.0	22.5	29.3

Cycle size is measured by instruction counts. Numbers presented in this table correspond to the largest data set for each benchmark.

Table 10.2: Comparison average cycle size of sequential and parallel versions

10.3 Directions for future research

Lazier collection-oriented match: Consider the task of finding all nodes in a network that are three hops away from a given node. Figure 10.1 shows a production that implements this. Under tuple-oriented semantics, one instantiation is generated for every path of length three from the root node. For the network in Figure 10.2, ten tuple-oriented instantiations are generated, one each for A and D and three each for B and C. Since the production conditions discriminate between every such path, collection-oriented match would generate the same set of instantiations. However, the operations performed by the production need only the leaf nodes and not the paths to these nodes. Since collection-oriented match guarantees the mutual consistency between all tuples in a collection-oriented instantiation, it is unable to collapse all the paths to a node. A variation of collection-oriented match that relaxes the mutual consistency guarantee would be able to deal with such situations by extending the laziness to generate the paths only if needed. The key component of such an approach would be an algorithm that analyzes the productions and determines the conditions for which the mutual consistency guarantee could be relaxed.

```
(p find-three-deep-leaves
  (root-node ^id <root>)
  (node ^id <intermediate1> ^parent <root>)
  (node ^id <intermediate2> ^parent <intermediate1>)
  (node ^id <leaf> ^parent <intermediate2>)
  -->
  (do-something <leaf>))
```

Figure 10.1: Production that finds leaves two hops away from the root

Optimizations for collection-oriented languages: The current implementation of COPL is a first-cut implementation and is relatively under-optimized. A host of optimizations have been discussed and proposed but are yet to be implemented. These include:

- *Sharing of collections between tokens:* Currently, collections in a parent token are copied whenever a successor is created. This is usually not necessary and it is possible to statically determine when it and when it is not needed. For large collections, avoiding copying collections could make a significant difference.
- *Using equivalence classes as collections:* For productions that contain no negated conditions, it is possible to use the equivalence classes directly as collections. This can reduce

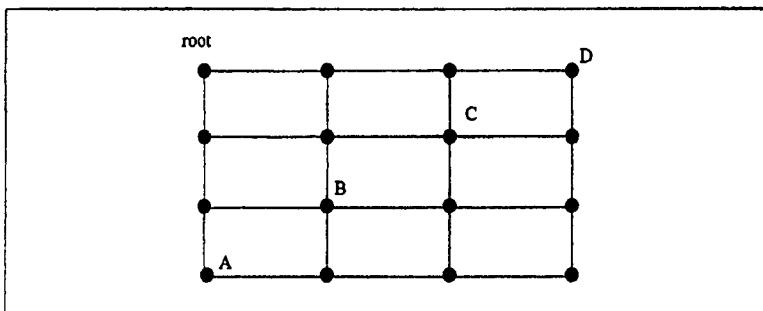


Figure 10.2: Example network

the cost of matching a new tuple to the cost of finding the appropriate equivalence class and adding the tuple to it. This optimization avoids performing β tests for tuples that join pre-existing equivalence classes. Since β tests are by far the most expensive part of the match process, this optimization has potential for large speedups for the programs it is applicable to. With the availability of collection-oriented operations, negated conditions are expected to be rare (see Section 8.5.2 for details).

- *Hashing right memories:* As the number of equivalence classes grows, it is beneficial to organize a right memory into a hash table of equivalence classes. In combination with the previous optimization, this could reduce the cost of adding a tuple, under appropriate circumstances, to a few instructions.

Parallelization of collection-oriented languages: One of the conclusions presented in the previous section was that collection-oriented production languages and match algorithms can be expected to alleviate or eliminate the primary limitations on speedups in production system programs. This is a qualitative conclusion. For a quantitative characterization of the effect of collection-oriented match algorithms and languages, an empirical study of a parallel implementation of a collection-oriented language is required. Some of the issues that need to be addressed are:

- *Load balancing for collection-oriented tokens:* collection-oriented tokens are essentially a grouping of a collection of tuple-oriented tokens. To process a collection-oriented token, it is necessary to compare a sequence of collections of tuples (the token) with a collection of equivalence classes (the right memory). Since collection-oriented tokens could vary greatly in size, a different task decomposition will probably be required for adequate load-balancing.

- *Parallelism within actions:* Collection-oriented production system languages allow the programmer to generate and manipulate collections of tuples and values. In particular, they allow her to move aggregate operations which cannot be parallelized within the production system paradigm to other languages where they can. Examples of such operations include sorting, accumulations and permutations.

Axiomatic specification of tuple-space contents: As mentioned in Section 3.2, the primary decision to be made in the design of a parallel production system language is the manner in which information about the contents of the tuple-space is to be provided. An obvious way to allow the programmer to specify this information would be to provide a data description sublanguage. Such a language could specify, for example, axioms about sets embedded in the flat tuple-space, their cardinality, their relationship with other sets and so forth. The main advantage of this approach is its explicit nature. It would make explicit the assumptions with which the program is parallelized. At the cost of greatly slowing down the execution, it is possible to check if these assumptions hold throughout the execution. Such a checker would be a useful debugging and maintenance tool. This approach was not adopted in this investigation because it was believed that the existence of an explicit operational model facilitates performance tuning. Since one of the primary goals of the investigation was to demonstrate that there is no program-independent bound on the speedups achievable in production system programs, an approach that makes it easy to extract the highest possible speedup was preferred. However, data specification sublanguages remain a viable and interesting avenue for future work.

Evaluation of performance on non-uniform memory access machines: The parallelism results presented in this dissertation were performed on a simulator which assumed a uniform memory access model. The results presented in Section 5.3.7 show that the parallelization overhead associated with the fine-grain decomposition can swamp a small cardinality multiprocessor. To achieve good speedups on multiprocessors containing 4-16 processors, coarse-grain decompositions are necessary. Section 5.3.7 presents results for a coarse-grain decomposition for one of the benchmarks. It would be interesting to find out if such decompositions are possible for other production system programs and if so, how much speedup can be achieved for them.

Appendix A

Trace formats used in the parallelism experiments

There are two trace formats used. The static trace file contains compile-time information about the program being traced. It is generated by the `pplc` compiler and is read in by the simulator at startup time. The dynamic trace file contains information from program execution. The two traces complement each other. The following sections describe the formats of both the trace files.

A.1 Static trace

The high level organization of the static trace file is:

1. Version number
2. Number of production-sets in the program
3. Information about individual production-sets
4. Number of Rete network nodes in the program
5. Information about individual nodes
6. Number of external functions called from the program
7. Information about individual functions

The following subsection provide details about individual records used in the static trace.

A.1.1 Information about individual production-sets

The trace record for a production-set consists of a header record followed by records for individual productions contained in the production-set. The trace record for a production consists of information about the production followed by a sequence of records for the actions in the production. The trace record for an action record consists of information about the type and number of operations per action. The C structures corresponding to the individual records are:

```
typedef struct TRACE_PSET_INFO_REC
{
    char *name;
    int num_productions;
} trace_pset_info_rec, *trace_pset_info_ptr;

typedef struct TRACE_PRODUCTION_INFO_REC
{
    char *name;
    int size;
    int positive_ce_count;
    int test_count;
    int num_of_actions;
    int is_parallel;
} trace_production_info_rec, *trace_production_info_ptr;

typedef struct TRACE_ACTION_INFO_REC
{
    int num_const_value_items;
    int num_variable_value_items;
    int num_function_value_items;
} trace_action_info_rec, *trace_action_info_ptr;
```

A.1.2 Information about individual functions

```
typedef struct TRACE_FUNCTION_INFO_REC
{
    char *name;
    int num_of_args;
} trace_function_info_rec, *trace_function_info_ptr;
```

A.1.3 Information about individual nodes

```
typedef struct TRACE_NODE_INFO_REC
{
    /* bit 0 contains node type -- AND/NOT, bits 1-31 contain depth (level)
     * of the node in the Rete network */
    int level_and_type;
    int num_of_successors;
    int num_of_tests;
    trace_test_info_ptr tests;
} trace_node_info_rec, *trace_node_info_ptr;

typedef struct TRACE_TEST_INFO_REC
{
    int test; /* the test being applied */
    int type; /* type of the values being tested */
} trace_test_info_rec, *trace_test_info_ptr;

/* tests used */
#define TRACE_EQ          0
#define TRACE_GT          1
#define TRACE_GE          2
#define TRACE_LT          3
#define TRACE_LE          4
#define TRACE_NE          5

/* runtime type tags */
#define TRACE_TYPE_INT    0x0
#define TRACE_TYPE_SYM    0x1
#define TRACE_TYPE_REAL   0x2
#define TRACE_TYPE_MISC   0x3
```

A.2 Dynamic trace

The high-level organization of the dynamic trace is:

1. version number
2. first match-select-act cycle traced
3. algorithm used for deletion of tuples

4. size of the hashtable used for implementing memory nodes
5. whether global or local hashtables were used
6. whether the hashtable can be grown dynamically
7. magic primary number used to randomize node identifiers
8. whether the trace was generated from a uniprocessor
9. fields specific to multiprocessor traces
 - (a) number of processors
 - (b) number of task pools
 - (c) whether tasks are to be grouped
 - (d) task pools management scheme - stacks or queues
 - (e) whether the tasks are initially distributed over task pools
 - (f) whether different actions in an instantiation can be done in parallel
10. individual trace records

The trace records are organized in segments, each segment corresponding to one match-select-act cycle. The first record in each segment is a `resolve` record. The final record in the trace is a `resolve` record with a special cycle number. Each schedulable task that can lead to the generation of other tasks is assigned an *activation id* which is used to indicate dependency information. The trace does not contain pointers.

```
typedef struct PPL_TRACE_RECORD_REC
{
    int proc_num_and_record_type; /* b_0-15 for type, rest for proc */
    union
    {
        struct
        { /* record of operations for each alpha activation */
            int tuple_time_tag;
            int size;
            int activation_id;
            int num_of_tests; /* currently ignoring diffs between tests */
            /* bit_0 = add_or_delete, bit_1-31 = num_of_successors */
            int num_of_successors_and_add_or_delete;
        } alpha;
    };
}
```

```
struct
{ /* record of operations for each beta activation */
    int left_parent_id;
    int right_parent_id;
    int activation_id;
    int node_id; /* used to find tests for node in static trace */
    /* bit_0=add/delete,bit_1=left/right, bit2-3 for scheduler,
     * bit_4-31 num_of_successors */
    int attributes;
    int *test_values; /* does not support fltpt values in trace */
} beta;
struct
{ /* record of operations for each pnode activation */
    int left_parent_id;
    int right_parent_id;
    /* bit_0 = add_or_delete, bit_1-31 = thread_id */
    int thread_id_and_add_or_delete;
    int prod_id;
    int *timetags;
} pnode;
struct
{ /* this is the header record for a resolve-match-act cycle */
    int cycle_num;
    int record_id; /* other record ids are derived from this one */
} resolve;
struct
{
    int activation_id;
    int thread_id;
    int prod_id;
} fire;
struct
{ /* info about most actions is obtained from the static trace.
   * this trace contains information only about actions that
   * do a variable number of operations */
    int parent_id; /* id of fire record that caused this action */
    int type;
    union
    {
        struct { int timetag; } make;
        struct { int timetag_new, timetag_old; } modify;
        struct { int timetag_new, timetag_old; } copy;
    }
}
```

```
    struct { int timetag; } remove;
    struct { long cost; } call;
} info;
} action;
struct
{ /* operations for computing individual values in an action */
    int type;
    union
    {
        struct { int num_of_values; } substr_call;
        struct { long cost; } function_call;
    } info;
} value_item;
struct
{ /* for makes done from external routines */
    int parent_id; /* special id for first cycle and loads */
    int timetag;
} external_make;
} info;
} ppl_trace_record_rec, *ppl_trace_record_ptr;
```

Appendix B

Cost model used in parallelism experiments

This section lists both the items in the cost model as well as the values used for the experiments described in this dissertation. These costs are very closely associated with the simulator. To fully understand the various costs covered, it would be helpful to be conversant with either the PPL implementation or the simulator or both.

Total number of cost model items:	277
Match-select-act cycle costs :	
Cost of straight line multiproc code in <i>msa</i> cycle:	25
Cost of straight line uniproc code in <i>msa</i> cycle:	25
Cost of finalizing <i>msa</i> cycle routine:	12
Left memory processing costs :	
Fixed uniproc cost:	21
Fixed multiproc cost:	34
Cost of initializing search loop:	9
Cost of testing node id:	4
Cost of initializing test loop	5
Cost of test loop body:	5
Cost of test loop update:	3
Cost of testing a word.	2
Cost of testing a tuple:	4
Cost of breaking out of test loop	2
Cost of testing a tuple pair:	5
Cost of search loop update:	5
Cost of deleting intermediate token:	4
Cost of deleting first token:	6
Cost of freeing conjugate β cell:	10

Cost of freeing non-conjugate β cell.	5
Fixed cost of freeing a β cell:	3
Cost of checking for an empty bucket:	1
Fixed cost of making a cell:	5
Cost of initializing cell creation loop:	4
Cost of cell creation loop body:	6
Cost of checking node depth:	3
Cost of testing a tuple pointer in a token:	4
Cost of testing for null in a token:	2
Cost of setting conjugate bucket to null:	3
Cost of storing a tuple pointer in a token:	5
Cost of storing a tuple pointer in a conjugate token:	4
Cost of adding a token to main bucket:	9
Cost of adding a token to conjugate bucket:	12
Cost of adding token to a bucket (uniproc):	8
Cost of setting the return value:	1
Right memory costs :	
Fixed right memory cost for uniproc:	11
Fixed right memory cost for multiproc:	26
Cost of initializing the search loop:	7
Cost of testing the node id:	4
Cost of testing a tuple pointer:	4
Cost of search loop update:	5
Cost of testing a word:	2
Cost of storing intermediate tokens:	4
Cost of storing first token:	6
Cost of freeing a conjugate cell:	10
Cost of freeing a non-conjugate cell:	5
Cost of freeing a cell for a uniproc:	3
Cost of adding a token to a conjugate bucket:	17
Cost of adding a token to a non-conjugate bucket:	13
Tuple processing costs :	
Cost of adding a tuple (multiproc):	35
Cost of adding a tuple (uniproc):	20
Fixed cost of deleting a tuple (multiproc):	20
Fixed cost of deleting a tuple (uniproc):	12
Cost of actually deleting the tuple (multiproc):	15
Cost of actually deleting the tuple (uniproc):	12
Cost of setting tuple-space pointer (multiproc):	3
Cost of setting tuple-space pointer (uniproc):	1
Cost of setting a value:	2
Cost of calling delete_tuple():	4

Fixed cost of creating a tuple (multiproc):	50
Fixed cost of creating a tuple (multiproc):	38
Cost of allocating tuple from free list:	4
Cost of calling malloc (multiproc):	6
Cost of calling malloc (uniproc):	4
Cost of calling finalize.tuples():	9
Fixed cost of freeing a list of tuples (uniproc):	6
Fixed cost of freeing a list of tuples (multiproc):	15
Cost of loop body for freeing a tuple list (uniproc):	14
Cost of loop body for freeing a tuple list (multiproc):	20
Cost of finding the free list by free list map:	5
Cost of directly finding the free list:	2
Fixed cost of calling remove.tuple():	27
Cost per tuple deleted by remove.tuple():	10
Fixed cost of calling make.tuple():	58
Fixed cost of calling modify.tuple() (multiproc):	78
Fixed cost of calling modify.tuple() (uniproc):	67
Fixed cost of calling copy.tuple:	58
Cost of loop header of updating tuple fields:	16
Cost of loading field index:	1
Cost of loading field value:	5
Cost of loop header for tuple copying:	22
Cost of loop body for tuple copying:	7
Cost of handling each index in a list of indices:	19
Cost of initializing a tuple:	5
Cost of calling match.wme():	8
Token processing costs :	
Fixed cost of making a β cell:	30
Cost of allocating β cell from a free list:	4
Cost of calling ppL_malloc():	4
Fixed cost in deleting a list of β cell:	6
Cost of loop body in deleting a list of β cell:	14
Cost of getting the free list from the map:	3
Cost of getting the free list from a cache:	2
Cost of initializing loop for β activation with hashing:	7
Cost of initializing loop for β activation without hashing:	6
Cost of calling the memory node routine:	8
Cost of loop update in β activation routines:	4
Cost of testing the node id.	4
Cost of updating the reference count for left tokens.	4
Cost of testing reference count for a left token:	4
Cost of computing reference count for right token:	5

Cost of reversing direction for rightnot activations:	2
Cost of freeing lock if hashing can be constant-folded:	6
Cost of freeing lock if hashing cannot be constant-folded:	5
Cost of testing a cached first value in bucket:	6
Cost of testing a uncached first value in bucket:	12
Cost of testing a cached non-first value in bucket	4
Cost of testing a uncached non-first value in bucket:	10
Fixed cost of making an α token:	19
Cost of allocating from a free list:	5
Cost of calling <code>ppl_malloc()</code> :	3
Fixed cost of freeing a list of α tokens:	6
Freeing cost per α token:	9
Fixed cost of α test:	28
Cost per α test per tuple:	4
Cost of callee-save per data-item for and:	3
Cost of callee-save per data-item for leftnot:	1
Cost of callee-save per data-item for rightnot:	5
Cost of callee-save per data-item for and (multiproc):	1
Cost of callee-save for final data-item:	1
Cost of caching a left value:	2
Cost of caching a right value:	1
Cost of calling the memory reclaiming routine for tokens:	13
Cost of caching the hash value:	1
Pnode processing costs :	
Cost of calling a pnode routine:	10
Fixed cost of processing pnode (multiproc):	40
Fixed cost of processing pnode (uniproc):	31
Cost of deleting instantiation from list (multiproc):	12
Cost of deleting instantiation from list (uniproc):	7
Cost of calling instantiation searching routine:	8
Cost of calling routine to remove instantiation:	6
Cost of calling routine to add instantiation:	11
Cost of consing an instantiation to list:	13
Conflict set costs :	
Constant cost for select:	26
Cost per production set (loop header):	14
Cost of checking if a production set has an instantiation:	32
Constant cost per parallel production (uniproc):	7
Constant cost per parallel production (multiproc):	8
Cost per instantiation:	20
Cost of checking if instantiation set is empty (uniproc):	30
Cost of checking if instantiation set is empty (multiproc):	35

Cost of checking if there is a new top instantiation.	20
Fixed cost of creating the second level heap:	24
Fixed cost per iteration of heap creation loop:	18
Cost of exchanging production sets:	28
Cost of updating indices to production sets:	1
Cost of comparing siblings:	15
Fixed cost of inserting an production set into the heap.	37
Fixed cost per iteration of insertion loop:	13
Cost of exchanging production sets for insertion:	19
Instantiation list processing costs :	
Cost of adding to an instantiation list.	23
Constant cost of deleting from an instantiation list:	26
Cost of hashing for an instantiation list:	11
Cost of delete loop header for instantiation lists:	3
Cost of testing if previous instantiation exists:	2
Cost of loop updating for deletion routine:	12
Cost of deletion for last instantiation in list:	11
Cost of deletion for intermediate instantiation in list:	10
Instantiation set processing costs :	
Cost of adding an instantiation set:	30
Constant cost for deleting an instantiation set from heap:	22
Cost of deleting last set:	2
Constant cost of deleting intermediate set:	35
Per set cost of deleting intermediate set:	21
Cost of exchanging instantiation sets for deletion:	24
Fixed cost of searching for an instantiation:	26
Fixed cost of instantiation searching loop:	8
Cost of returning from the middle of the search loop:	3
Cost of search loop update:	5
Fixed cost of adding an instantiation (multiproc):	49
Fixed cost of adding an instantiation (uniproc):	46
Cost of loop test for adding an instantiation:	10
Cost of loop body for adding an instantiation:	19
Cost of determining if the top element has changed:	7
Cost of handling new top element (multiproc):	16
Cost of handling new top element (uniproc):	8
Task record processing costs :	
Cost of creating a task record from free list:	5
Cost of calling ppl_malloc():	3
Cost of creating a match task:	26
Cost of creating a firing task:	20
Cost of scheduling a firing task:	29

Cost of scheduling a match task:	29
Cost of completing scheduling for match task :	4
Cost of completing scheduling for firing task :	4
Cost of waking up processes:	18
Cost of loop header in scheduler	12
Cost of unrolled iteration init in scheduler:	13
Cost of rolled iteration init in scheduler:	15
Cost of jumping to top of loop:	5
Cost of extracting a task record from a task pool:	11
Cost of setting the activity bitmap:	12
Common cost of running a task:	7
Extra cost for running a match task:	6
Extra cost of running a firing task:	7
Cost of freeing a task:	7
Cost of scheduling the root node for uniproc.	11
Cost of scheduling the root node for multiproc:	12
Firing costs :	
Cost of first block in multiproc case:	24
Cost of second block in uniproc:	8
Fixed cost (uniproc):	8
Cost of loop body (multiproc):	18
Cost of loop body (uniproc):	19
Cost of fire loop update.	4
Cost of final block for firing.	5
Fixed cost of dollar.assert():	12
Fixed cost of actions:	15
Cost per action for rhs:	2
Cost per variable argument:	4
Cost to get the class size:	16
Cost of accessing a symbol given its id:	33
Cost of substr with known bounds:	42
Cost of substr with unknown vars:	10
Instantiation processing costs :	
Fixed cost of comparing instantiations of different productions:	37
Cost of loading value (diff prods) for comparison:	1
Cost of a single test (diff prods) for comparison:	6
Cost of update for comparison loop (diff prods):	4
Fixed cost of comparing instantiations of the same production:	22
Cost of a single test (same prod) for comparison:	6
Cost of update for comparison loop (same prod):	5
Fixed cost of deleting an instantiation (multiproc):	70
Fixed cost of deleting an instantiation (uniproc):	66

Cost of locking for deleting an instantiation	10
Cost of header for loop arranging upper heap:	13
Cost of exchange for upper heap loop:	28
Cost of index increment for upper loop:	1
Cost of loop test for upper loop:	10
Cost of header for loop arranging lower heap:	12
Cost of exchange for lower heap:	29
Cost of placing an instantiation on tmp free list:	3
Cost of updating tmp free list:	4
Fixed cost of freeing tmp free list:	10
Cost of loop body for freeing tmp free list:	29
Cost of mapping size to free list:	3
Cost of dereferencing ptr to get free list:	2
Fixed cost of hashing full instantiation:	22
Cost of hashing each timetag in instantiation:	6
Fixed cost of hashing instantiation in parts:	4
Cost of getting first timetag for hashing by parts:	3
Cost of loop header for hashing by parts:	9
Cost of loop update for hashing by parts:	8
Cost of extracting a timetag for hashing by parts:	4
Cost of extracting a null value for hashing by parts:	1
Cost of extracting last timetag for hashing by parts:	5
Fixed cost of ordering timetags:	47
Cost of header for first timetag ordering loop:	6
Cost of header for second timetag ordering loop:	11
Fixed cost for creating timetag heap:	12
Cost of testing a ptr for heap creation:	6
Cost of updating a ptr for heap creation:	2
Cost of comparing two ptrs for heap creation:	6
Cost of update in heap creation loop:	10
Cost of main routine for instantiation deletion (uniproc):	15
Cost of main routine for instantiation deletion (multiproc):	18
Cost of main routine for instantiation addition (uniproc):	13
Cost of main routine for instantiation addition (multiproc):	27
Fixed cost of creating an instantiation:	53
Cost of allocating instantiation record from free list:	4
Cost of calling ppl_malloc() for record:	7
Cost of copying first tuple ptr:	7
Fixed cost of token creation loop:	9
Cost of getting a null tuple ptr:	12
Cost of getting a non-null tuple ptr:	16
Cost of getting a null last tuple:	3

Cost of getting a non-null last tuple:	7
Cost of loop update:	10
Fixed cost of matching instantiations:	5
Cost of checking for single instantiation:	7
Cost of checking for null last tuple:	2
Cost of matching a null tuple:	7
Cost of matching a full tuple:	8
Cost of match loop header:	6
Fixed loop body for match loop:	5
Cost of update for match loop:	5
List consing costs :	
Cost of straight line code:	18
Cost of allocation from free list:	5
Cost of calling malloc for allocation:	3
Cost of consing up a singleton list:	20
Free list processing costs :	
Cost of non-loop code:	5
Cost of each loop iteration:	9
Memory allocation costs :	
Constant uniproc cost for memory allocation:	35
Constant multiproc cost for memory allocation:	40
Cost of adjusting size of request:	4
Cost of mapping to size range:	10
Cost of double word aligned allocation (uniproc):	22
Cost of double word aligned allocation (multiproc):	27
Costs for calling sequence of routines :	
Fixed cost per call:	1
Additional cost per argument:	1
Locking routines :	
Cost of acquiring a lock:	10
Cost of releasing a lock:	3
Cost of creating and initializing a lock	8

Appendix C

Configuration file for the simulator

The configuration file controls the simulator and makes it easier to run a sequence of simulations without human intervention. This appendix describes the parameters that can be specified in a configuration file. The simulator has default values for all these parameters. It also accepts command-line arguments to set all these parameters. The values specified in a configuration file override default values and values specified on the command-line override the values in the configuration file.

Program name : name of the program being simulated. This is used to create the trace file name (or the socket name if the trace is being sent to a socket). It is also used to create default names for the static and dynamic trace files as well as the output file.

Static trace file name : overrides the default name generated using the program name. It is always a file unlike dynamic trace file which can be a Unix or inet socket.

Cost model file name : the simulator is not tied into any particular processor. It reads in the cost model which is specified as a sequence of (string,integer) pairs.

Dynamic trace file name : this can be a file or a socket. Must be readable by the user.

Output file name : overrides the default generated from the program name. Allows multiple simulations to run in parallel.

First cycle to be simulated : useful for partial simulations.

Last cycle to be simulated : again, useful for partial simulations

Compute flags : these flags specify what information should be computed by the simulator.

- Overall runtime
- Maximum parallelism during the execution
- Maximum parallelism during each cycle

- Processor utilization
- User level information (e.g., number of firings *etc.*)
- Detailed breakdown of execution time
- Cost per cycle
- Number of tasks per cycle
- Number of tasks in the program
- Number of hashtable activations
- Number of conflict set activations

Verbose : controls printing of diagnostic details

Help : prints information about the options available

Number of processors : ignored for uniprocessor simulations

Number of task pools : ignored for uniprocessor simulations

Uniprocessor simulations : this flag makes all the multiprocessor related flag invalid

Appendix D

Detailed parallelism results

This appendix presents detailed results from the parallelism experiments. For every benchmark, it presents three groups of results – language level, task breakdown and execution time breakdown. The language level results consist of information that is usually available to the programmer, for example the number of instantiations fired. The other two are internal statistics from the runtime system. The task breakdown results also include a measure of the program efficiency, referred to as *efficiency of instantiation generation*. It is the fraction of the generated instantiations that are actually fired. Efficient programs have a high EIG. Programs that perform selection or accumulation within the production system model have a low EIG. The execution time breakdown is for uniprocessor execution, that is for the parallel-model versions of the programs. These versions do not include the cost for parallelization. It is expected that, in the absence of the parallelization costs (which depend on the architecture of the machine), these numbers reflect inherent characteristics of the programs. In the tables, execution time breakdown is presented in terms of percentage of total time. In addition to the seven explicit headings, there is an *other* heading which includes the cost of book-keeping in the *match-select-act* cycle and the cost of removing the selected instantiations from the conflict set. In the execution time breakdown statistics, all the times shown are non-overlapping except the time spent in actions and (external) functions. In this case, the time for actions includes the time for functions. The latter have been shown separately to illustrate the point that, for some programs, external function calls can dominate the entire computation.

Version	data set	\rightarrow	meffct1	dc46809	sf4917
Parallel	info 1				
	firings	67055	98116	244413	
	cycles	2288	3418	10052	
	tuple deletes	27046	39994	106680	
	tuple adds	30942	45877	112386	

Table D.1: Language level results for spam

Version	data set	\rightarrow	1	10	20	30	40	50	60	70
Sequential	info 1									
	firings	942	4812	9112	13412	17112	22012	26312	30612	
	cycles	944	4814	9114	13414	17114	22014	26314	30614	
	tuple deletes	1041	4911	9211	13511	17811	22111	26411	30711	
Parallel	tuple adds	1062	3067	9517	13967	18417	22867	27317	31767	
	firings	423	2358	4508	6658	8808	10958	13108	15258	
	cycles	225	360	510	660	810	960	1110	1260	
	tuple deletes	522	2457	4607	6757	8907	11087	13207	15357	
	tuple adds	543	2613	4913	7213	9513	11813	14113	16413	

Table D.2: Language level results for life

Version	data set	→	25	50	75	100	125	150	175	200	225	250	275
	info	↓											
Sequential	firings	4130	7952	11715	15495	19114	23131	26894	30663	34457	37922	41506	
	cycles	4131	7953	11716	15496	19115	23132	26895	30664	34458	37923	41507	
	tuple deletes	4229	8051	11814	15594	19213	23250	26993	30762	34556	38021	41605	
	tuple adds	4283	8155	11968	15798	19467	23534	27347	31166	35010	38525	42159	
	firings	2733	5261	7749	10247	12641	15297	17785	20275	22783	25069	27141	
	cycles	204	204	204	204	204	204	204	204	204	204	204	
Parallel	tuple deletes	2832	5360	7848	10346	12740	15296	17884	20374	22882	25168	27540	
	tuple adds	2886	5464	8002	10550	12994	15700	18238	20778	23336	25672	28094	

Table D.3: Language level results for circuit

Version	data set	→	10	20	30	40	50	60	80	90	100	110	120
	info	↓											
Sequential	firings	4964	9924	14884	19844	24804	29764	39684	44644	49604	545564	59574	
	cycles	4965	9925	14885	19845	24805	29765	39685	44645	49605	54565	59525	
	tuple deletes	3202	6402	9602	12802	16002	19202	25602	28802	32002	35202	38402	
	tuple adds	5272	10522	15772	21072	26272	31522	42022	47272	52522	57772	63022	
	firings	4964	9924	14884	19844	24804	29764	39684	44644	49604	545564	59574	
	cycles	32	32	32	32	32	32	32	32	32	32	32	
Parallel	tuple deletes	3202	6402	9602	12802	16002	19202	25602	28802	32002	35202	38402	
	tuple adds	5272	10522	15772	21072	26272	31522	42022	47272	52522	57772	63022	

Table D.4: Language level results for waltz

Version	data set → info 1	1	2	3	4	5	6	7	8	9	10
Sequential	firings	1903	2873	3843	4813	5783	6753	7723	8693	9663	10633
	cycles	1905	2875	3845	4815	5785	6755	7725	8695	9665	10635
	tuple deletes	4022	5842	7662	9482	11302	13122	14942	16762	18582	20402
	tuple adds	6144	9270	12396	15522	18648	21774	24900	28026	31152	34278
Parallel	firings	1961	2966	3971	4976	5981	6986	7991	8996	10001	11006
	cycles	110	110	145	190	235	280	325	370	415	460
	tuple deletes	4102	5992	7882	9772	11662	13552	15442	17332	19222	21112
	tuple adds	6237	9433	12629	15825	19021	22217	25413	28609	31805	35001

Table D.5: Language level results for hotel

Version	data set info	→	moffett1	dc36809	sf4917
Parallel	α tasks	57988	85781	219066	
	add β tasks	59191	82968	135581	
	delete β tasks	17641	24253	43191	
	add pnode tasks	70933	103634	257279	
	delete pnode tasks	3873	5518	12845	
	total tasks	343268	498281	1153869	
	EIG (%)	94.53	95.68	95.00	

Table D.6: Task breakdown for spam

Version	data set →	1	10	20	30	40	50	60	70
Sequential									
	α tasks	2105	9978	18728	27478	36228	44978	53728	62478
	add β tasks (million)	0.028	0.224	0.441	0.658	0.875	1.092	1.309	1.526
	delete β tasks (million)	0.028	0.219	0.432	0.645	0.858	1.070	1.283	1.495
	add probe tasks	1142	5012	9312	13612	17912	22212	26512	30812
	delete probe tasks	200	200	200	200	200	200	200	200
	EIG (%)	82.49	96.01	97.85	98.53	98.88	99.10	99.25	99.35
Parallel									
	α tasks	1065	5070	9520	13970	18420	22870	27320	31770
	add β tasks (million)	0.027	0.244	0.485	0.726	0.967	1.208	1.449	1.690
	delete β tasks (million)	0.026	0.239	0.476	0.713	0.950	1.187	1.424	1.661
	add probe tasks	823	4558	8708	12858	17008	21158	25308	29458
	delete probe tasks	400	2200	4200	6200	8200	10200	12200	14200
	total tasks (million)	0.056	0.499	0.992	1.485	1.978	2.471	2.964	3.457
	EIG (%)	51.40	51.73	51.77	51.78	51.79	51.79	51.80	

Table D.7: Task breakdown for life

Version	data set → info	25	50	75	100	125	150	175	200	225	250	275
Sequential	α tasks	8312	16206	23782	31392	38680	46764	54340	61928	69566	76546	83764
	add β tasks (million)	0.191	0.371	0.548	0.727	0.898	1.066	1.234	1.441	1.620	1.785	1.955
	delete β tasks (million)	0.189	0.368	0.544	0.721	0.891	1.078	1.254	1.430	1.607	1.771	1.940
	add pnode tasks	4230	8032	11815	15595	19214	23231	26994	30763	34557	38022	41606
	delete pnode tasks	99	99	99	99	99	99	99	99	99	99	99
	EIG (%)	97.64	98.76	99.15	99.36	99.48	99.57	99.63	99.67	99.71	99.74	99.76
Parallel	α tasks	5718	10824	15830	20896	25734	31096	36122	41152	46218	50840	55634
	add β tasks (million)	0.1667	0.328	0.487	0.647	0.800	0.969	1.128	1.287	1.447	1.595	1.747
	delete β tasks (million)	0.165	0.325	0.482	0.641	0.793	0.956	1.117	1.275	1.433	1.580	1.731
	add pnode tasks	2908	5508	8076	10675	13149	15868	18442	21010	23582	25954	28430
	delete pnode tasks	175	247	327	428	508	571	657	735	799	885	989
	total tasks (million)	0.346	0.680	1.009	1.340	1.658	2.007	2.336	2.663	2.997	3.302	3.618
	EIG (%)	93.98	95.52	95.95	95.99	96.14	96.40	96.44	96.50	96.61	96.59	96.52

Table D.8: Task breakdown for circuit

Version	data set →	10	20	30	40	50	60	80	90	100	110	120
Sequential	info											
	α tasks	8474	16924	25374	33824	42274	50724	67624	76074	84524	92974	101424
	add β tasks (million)	0.034	0.068	0.102	0.136	0.171	0.205	0.273	0.307	0.341	0.375	0.409
	delete β tasks (million)	0.076	0.052	0.078	0.104	0.131	0.157	0.209	0.235	0.261	0.287	0.313
	add pnode tasks	5344	10684	16024	21364	26704	32044	42724	48064	53404	58744	64084
	delete pnode tasks	380	760	1140	1520	1900	2280	3040	3420	3800	4180	4560
	BIG (%)	92.89	92.89	92.89	92.88	92.88	92.88	92.88	92.88	92.88	92.88	92.88
Parallel	info											
	α tasks	8474	16924	25374	33824	42274	50724	67624	76074	84524	92974	101424
	add β tasks (million)	0.032	0.063	0.095	0.127	0.158	0.190	0.253	0.285	0.316	0.348	0.379
	delete β tasks (million)	0.024	0.048	0.072	0.106	0.120	0.144	0.193	0.217	0.241	0.265	0.289
	add pnode tasks	5384	10764	16144	21524	26904	32284	43044	48424	53804	59184	64564
	delete pnode tasks	420	840	1260	1680	2100	2520	3360	3780	4200	4620	5040
	total tasks (million)	0.079	0.157	0.236	0.314	0.393	0.471	0.628	0.707	0.786	0.864	0.943
	BIG (%)	92.20	92.20	92.20	92.19	92.19	92.19	92.19	92.19	92.19	92.19	92.19

Table D.9: Task breakdown for waltz

Version	data set →	1	2	3	4	5	6	7	8	9	10
Sequential	α tasks	10166	15112	20058	25004	29950	34896	39842	44788	49734	54680
	add β tasks (million)	0.022	0.043	0.068	0.098	0.133	0.173	0.219	0.269	0.324	0.384
	delete β tasks (million)	0.016	0.032	0.053	0.080	0.111	0.147	0.188	0.234	0.285	0.341
	add pnode tasks	4156	7911	12786	18601	25416	33231	42046	51861	62676	74491
	delete pnode tasks	2253	5098	8943	13788	19633	26478	34323	43168	53013	63858
	EIG (%)	45.79	36.04	30.06	25.87	22.75	20.32	18.37	16.76	15.42	14.27
Parallel	α tasks	10339	15425	20511	25597	30683	35769	40855	45941	51027	56113
	add β tasks (million)	0.021	0.039	0.061	0.086	0.116	0.149	0.186	0.226	0.271	0.319
	delete β tasks (million)	0.014	0.028	0.046	0.067	0.092	0.121	0.154	0.191	0.231	0.275
	add pnode tasks	14339	19155	25547	33539	43131	54323	67115	81507	97449	115091
	delete pnode tasks	12376	16189	21576	28563	37150	47337	59124	72511	84498	104085
	total tasks (million)	0.076	0.124	0.182	0.251	0.331	0.421	0.523	0.635	0.758	0.892
	EIG (%)	13.68	15.48	15.54	14.84	13.87	12.86	11.91	11.04	10.26	9.56

Table D.10: Task breakdown for hotel

Version	data	set	→	moffett	dc36809	st4917
	info	info	info	info	info	info
Parallel						
	α tasks	0.11	0.1	0.15		
	β tasks	0.09	0.08	0.08		
	proto tasks	0.45	0.47	1.09		
	actions	99.21	99.21	98.44		
	functions	99.06	99.07	98.24		
	mem reclaim	0.02	0.02	0.02		
	fire	0.00	0.00	0.00		
	others	0.12	0.12	0.22		

Table D.11: Execution time breakdown for spam

Version	data set →	1	10	20	30	40	50	60	70
	info								
Sequential	α tasks	2.21	1.61	1.53	1.49	1.46	1.42	1.40	1.38
	β tasks	74.69	79.49	79.39	79.63	79.69	79.87	80.11	80.13
	phnode tasks	8.71	5.54	5.20	5.02	4.91	4.79	4.59	4.62
	actions	5.08	3.48	3.29	3.19	3.12	3.05	3.00	2.95
	functions	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	mem reclaim	5.77	5.35	5.21	5.10	5.01	4.91	4.83	4.76
	fire	0.09	0.06	0.05	0.05	0.05	0.05	0.05	0.05
	others	3.45	4.47	5.13	5.5	5.76	5.91	6.02	6.11
Parallel	α tasks	1.11	0.89	0.85	0.83	0.80	0.79	0.77	0.75
	β tasks	80.41	84.25	84.59	84.74	84.87	85.08	85.32	85.47
	phnode tasks	9.03	7.72	7.78	7.82	7.88	7.84	7.78	7.78
	actions	2.65	1.55	1.44	1.39	1.35	1.31	1.28	1.25
	functions	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	mem reclaim	5.24	5.10	4.85	4.85	4.74	4.63	4.53	4.43
	fire	0.08	0.07	0.06	0.06	0.06	0.06	0.06	0.06
	others	1.48	0.42	0.33	0.31	0.3	0.29	0.26	0.26

Table D1.12: Execution time breakdown for life

Version	data set	→	25	50	75	100	125	150	175	200	225	250	275
Sequential	info		4.85	4.84	4.81	4.78	4.76	4.73	4.70	4.67	4.65	4.63	4.61
	α tasks		71.47	70.48	69.83	69.37	69.06	68.81	68.63	68.36	68.38	68.29	68.20
	β tasks		7.68	7.64	7.67	7.70	7.71	7.70	7.70	7.68	7.69	7.68	7.67
	phonetasks		4.66	4.54	4.48	4.44	4.41	4.37	4.34	4.31	4.29	4.27	4.25
	actions		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	functions		5.84	5.71	5.64	5.60	5.56	5.51	5.47	5.44	5.41	5.39	5.37
	mem reclaim		0.18	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08
	fire		5.42	6.71	7.49	8.03	8.42	8.8	9.08	9.26	9.5	9.66	9.82
	others		4.12	4.09	4.07	4.06	4.05	4.03	4.01	4.00	3.99	3.98	3.96
Parallel	α tasks		78.48	78.70	78.65	78.63	78.64	78.67	78.69	78.69	78.71	78.71	78.71
	β tasks		7.40	7.56	7.71	7.85	7.93	8.00	8.05	8.09	8.12	8.15	8.19
	phonetasks		3.26	3.14	3.09	3.07	3.05	3.03	3.01	3.00	2.99	2.97	2.96
	actions		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	functions		5.54	5.52	5.51	5.50	5.49	5.46	5.44	5.42	5.41	5.40	5.39
	mem reclaim		0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18
	fire		1.02	0.81	0.74	0.69	0.67	0.66	0.64	0.62	0.62	0.61	0.61
	others												

Table D.13: Execution time breakdown for circuit

Version	Data set	→	10	20	30	40	50	60	80	90	100	110	120
	info	↓											
Sequential	α tasks	2.21	2.06	1.94	1.90	1.86	1.85	1.77	1.72	1.71	1.71	1.71	1.71
	β tasks	39.63	41.39	43.53	43.98	44.50	44.37	45.93	47.15	47.31	47.04	46.91	46.91
	procedure tasks	16.34	15.50	14.74	14.49	14.26	14.21	13.69	13.20	13.12	13.15	13.15	13.15
	actions	12.37	11.54	10.88	10.63	10.41	10.33	9.89	9.64	9.56	9.56	9.55	9.55
	functions	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	mem reclaim	3.53	3.50	3.12	3.05	2.98	2.96	2.84	2.77	2.74	2.74	2.74	2.74
	fire	0.18	0.17	0.16	0.16	0.15	0.15	0.14	0.14	0.14	0.14	0.14	0.14
	others	25.74	26.04	25.63	25.79	25.84	26.13	25.73	25.38	25.42	25.66	25.8	25.8
Parallel	α tasks	2.74	2.37	2.13	1.97	1.82	1.73	1.57	1.51	1.43	1.38	1.32	1.32
	β tasks	47.80	47.54	46.90	44.89	43.81	41.32	37.49	36.03	34.56	33.18	31.80	31.80
	procedure tasks	28.16	31.68	34.45	37.90	40.23	43.52	48.76	50.80	52.91	54.75	56.63	56.63
	actions	15.62	13.52	12.14	11.21	10.39	9.87	8.95	8.57	8.16	7.86	7.53	7.53
	functions	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	mem reclaim	2.53	2.19	1.96	1.81	1.68	1.60	1.45	1.39	1.32	1.27	1.22	1.22
	fire	0.66	0.57	0.52	0.48	0.44	0.42	0.38	0.36	0.35	0.33	0.32	0.32
	others	2.49	2.13	1.9	1.74	1.63	1.54	1.4	1.34	1.27	1.23	1.18	1.18

Table D.14: Execution time breakdown for waltz

Version	data set	→	1	2	3	4	5	6	7	8	9	10
Sequential	info											
	α tasks	1.94	0.50	0.17	0.08	0.04	0.02	0.02	0.01	0.01	0.01	0.01
	β tasks	59.39	86.98	94.37	96.89	98.00	96.57	98.91	99.13	99.28	99.39	99.39
	phnode tasks	18.78	7.15	3.51	2.13	1.47	1.11	0.88	0.72	0.61	0.53	
	actions	9.42	2.35	0.80	0.36	0.19	0.11	0.07	0.05	0.03	0.03	
	functions	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
	mem reclaim	1.92	0.60	0.24	0.13	0.08	0.05	0.04	0.03	0.02	0.02	
	fire	0.06	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
	others	8.49	2.4	0.91	0.41	0.22	0.14	0.08	0.06	0.05	0.02	
Parallel	info											
	α tasks	1.37	0.56	0.20	0.09	0.05	0.03	0.02	0.01	0.01	0.01	
	β tasks	30.76	74.30	89.83	94.73	96.71	97.68	98.24	98.59	98.83	99.01	
	phnode tasks	57.49	21.09	8.51	4.32	2.90	2.09	1.61	1.31	1.10	0.94	
	actions	6.66	2.65	0.95	0.41	0.21	0.12	0.08	0.05	0.04	0.03	
	functions	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
	mem reclaim	1.83	0.73	0.29	0.14	0.08	0.05	0.04	0.03	0.02	0.02	
	fire	0.12	0.05	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
	others	1.77	0.62	0.2	0.11	0.05	0.03	0.01	0.01	0.00	0.00	

Table D.15: Execution time breakdown for hotel

Appendix E

Code for benchmarks used in parallelism experiments

This appendix contains the source code for four of the benchmarks used in the parallelism experiments – life, circuit, waltz and hotel. The code for spam is not available for public distribution. Two versions are presented for every program – one that uses the parallel constructs and the other that does not.

E.1 Sequential version of the game of life

```
; Sequential version of the game of life in PPL
; Each cell is represented as a tuple. It contains a field for its status
; and four fields for links to neighbors.
; The rules of life are:
; 1. if an organism has < 2 neighbors, it dies of loneliness
; 2. if it has 2 or 3 neighbors it lives
; 3. a new organism is born in a cell with three neighbors
; 4. if it has 4 neighbors, it dies of overcrowdedness.
;
; Version notes: this is an efficient sequential version. Due to the
; single-instantiation-firing assumption, direct atomic update of all
; modified cells not possible. Instead, atomicity is achieved by creating
; copies of tuples corresponding to modified cells *without* destroying
; the old ones. In OPS5, this can be done by (make (substr <tuple>) (mods)).
; In PPL, it is done using the copy primitive. At the end of the
; computation for each generation, the old copies are deleted and the new
; copies are installed in their place. This requires an extra flag on
; the cell tuples which indicates whether the tuple contains the original
; cell or a modified copy. The tuple corresponding to a cell is modified
```

```
, if an only if the state of the cell changes. The original version always
; modified the cell tuples whether or not the state of the cell changed.
; "*" signifies an alive organism, "." signifies a dead organism
;-----
; This version by: Anurag Acharya, acha@cs.cmu.edu
; original version: ftp.cs.columbia.edu:pub/prosys/prosys.tar.Z
;-----

;-----
; Tuple declarations
;-----

; coordinates are needed for printing, id is needed for linking
(literalize cell x-coordinate y-coordinate id status left right up down
modified)

; generation tuple is used for counters and limits -- the type field can
; be either "current" or "desired"
(literalize generation type value)

; flag tuple is used to control the sequencing -- values are print,
; print-cells, do-computation, set-modified-flags, completed-print,
; compute-cells and completed
(literalize flag value)

; format tuple is used to store user format preferences -- values are
; no and yes. yes prints the map every generation, no prints
; the map only at the end of the simulation.
(literalize format value)

; this is used for traversing the grid for printing purposes
(literalize print-coordinates x-coordinate y-coordinate)

;-----
; Productions
;-----

(p start
  (start)
  -->
  (make generation current 0)
  (write "Generations to run simulations for: ")
  (make generation desired (accept))
  (write "Print after every generation ? ")
  (make format (accept))
  (make flag compute-cells))

(p switch-to-next-generation
```

```
((flag completed-print) <flag>)
  (generation ^type desired ^value <x>)
  ((generation ^type current ^value <y> < <x>)) <cur-gen>
-->
  (modify <cur-gen> ^value (compute <y> + 1))
  (modify <flag> compute-cells)

(p finished-simulation
  ((flag completed-print) <flag>)
  (generation ^type desired ^value <x>)
  (generation ^type current ^value <x>)
-->
  (modify <flag> completed))

; of the next three productions, only one fires in a given cycle. If the
; "per-generation" production matches, it will be selected due to
; specificity, else the vanilla "switch-to" production will fire.

(p switch-to-print
  ((flag print-cells) <flag>)
  (generation ^type desired ^value <x>)
  (generation ^type current ^value < <x>)
-->
  (modify <flag> print))

(p switch-to-print-final
  ((flag print-cells) <flag>)
  (generation ^type desired ^value <x>)
  (generation ^type current ^value <x>)
-->
  (modify <flag> print))

(p switch-to-print-per-generation
  ((flag print-cells) <flag>)
  (generation ^type desired ^value <x>)
  (generation ^type current ^value < <x>)
  (format ^value no)
-->
  (modify <flag> completed-print))

(p init-print
  (flag print)
  (generation ^type current ^value <x>)
-->
  (make print-coordinates ^x-coordinate 0 ^y-coordinate 0)
  (write generation <x> ":\n"))

(p print-next-cell-in-row
  ((print-coordinates ^x-coordinate <x> ^y-coordinate <y>) <coords>)
```

```

(cell ^x-coordinate <x> ^y-coordinate <y> ^status <status>)
-->
(write <status>)
(modify <coords> ^x-coordinate (compute <x> + 1))

(p switch-rows
  { (print-coordinates ^x-coordinate <x> ^y-coordinate <y>) <coords> }
  -(cell ^x-coordinate <x>)
-->
  (modify <coords> ^x-coordinate 0 ^y-coordinate (compute <y> + 1))
  (write (crlf)))

(p finalize-print
  { (flag print) <flag> }
  { (print-coordinates ^x-coordinate <x> ^y-coordinate <y>) <coords> }
  -(cell ^y-coordinate <y>)
-->
  (modify <flag> completed-print)
  {remove <coords>}
  (write "\n\n"))

(p implement-kill-rule-0
  { flag do-computation }
  { generation ^type desired ^value <x> }
  { generation ^type current ^value < <x> }
  { (cell ^status |.) ^left <left> ^right <right> ^up <up>
    ^down <down> ^modified no } <cell>
  { cell ^id <left> ^status |.| ^modified no }
  { cell ^id <right> ^status |.| ^modified no }
  { cell ^id <up> ^status |.| ^modified no }
  { cell ^id <down> ^status | | ^modified no }
-->
  {copy <cell> ^status |.| ^modified yes})

(p implement-kill-rule-left
  { flag do-computation }
  { generation ^type desired ^value <x> }
  { generation ^type current ^value < <x> }
  { (cell ^status |.) ^left <left> ^right <right> ^up <up>
    ^down <down> ^modified no } <cell>
  { cell ^id <left> ^status |.| ^modified no }
  { cell ^id <right> ^status |.| ^modified no }
  { cell ^id <up> ^status |.| ^modified no }
  { cell ^id <down> ^status | | ^modified no }
-->
  {copy <cell> ^status |.| ^modified yes})

(p implement-kill-rule-right

```

```

(flag do-computation)
(generation ^type desired ^value <x>)
(generation ^type current ^value <x>)
{((cell ^status [*] ^left <left> ^right <right> ^up <up>
^down <down> ^modified no) <cell>)
 (cell ^id <left> ^status [.| ^modified no)
 (cell ^id <right> ^status [*| ^modified no)
 (cell ^id <up> ^status [.| ^modified no)
 (cell ^id <down> ^status [.| ^modified no)
-->
(copy <cell> ^status [.| ^modified yes))

(p implement-kill-rule-up
(flag do-computation)
(generation ^type desired ^value <x>)
(generation ^type current ^value <x>)
{((cell ^status [*] ^left <left> ^right <right> ^up <up>
^down <down> ^modified no) <cell>)
 (cell ^id <left> ^status [.| ^modified no)
 (cell ^id <right> ^status [.| ^modified no)
 (cell ^id <up> ^status [*| ^modified no)
 (cell ^id <down> ^status [.| ^modified no)
-->
(copy <cell> ^status [.| ^modified yes))

(p implement-kill-rule-down
(flag do-computation)
(generation ^type desired ^value <x>)
(generation ^type current ^value <x>)
{((cell ^status [*] ^left <left> ^right <right> ^up <up>
^down <down> ^modified no) <cell>)
 (cell ^id <left> ^status [.| ^modified no)
 (cell ^id <right> ^status [.| ^modified no)
 (cell ^id <up> ^status [.| ^modified no)
 (cell ^id <down> ^status [*| ^modified no)
-->
(copy <cell> ^status [.| ^modified yes))

(p implement-birth-rule-left-right-up
(flag do-computation)
(generation ^type desired ^value <x>)
(generation ^type current ^value <x>)
{((cell ^status [.| ^left <left> ^right <right> ^up <up>
^down <down> ^modified no) <cell>)
 (cell ^id <left> ^status [*| ^modified no)
 (cell ^id <right> ^status [*| ^modified no)
 (cell ^id <up> ^status [*| ^modified no)
 (cell ^id <down> ^status [.| ^modified no)

```

```

-->
(copy <cell> ^status [*| ^modified yes))

(p implement-birth-rule-left-right-down
  (flag do-computation)
  (generation ^type desired ^value <x>)
  (generation ^type current ^value < <x>)
  {((cell ^status | | ^left <left> ^right <right> ^up <up>
  ^down <down> ^modified no) <cell>)
   (cell ^id <left> ^status [*| ^modified no)
   (cell ^id <right> ^status [*| ^modified no)
   (cell ^id <up> ^status | | ^modified no)
   (cell ^id <down> ^status [*| ^modified no)
  -->
  (copy <cell> ^status [*| ^modified yes))

(p implement-birth-rule-left-up-down
  (flag do-computation)
  (generation ^type desired ^value <x>)
  (generation ^type current ^value < <x>)
  {((cell ^status |..| ^left <left> ^right <right> ^up <up>
  ^down <down> ^modified no) <cell>)
   (cell ^id <left> ^status [*| ^modified no)
   (cell ^id <right> ^status |..| ^modified no)
   (cell ^id <up> ^status [*| ^modified no)
   (cell ^id <down> ^status [*| ^modified no)
  -->
  (copy <cell> ^status [*| ^modified yes))

(p implement-birth-rule-right-up-down
  (flag do-computation)
  (generation ^type desired ^value <x>)
  (generation ^type current ^value < <x>)
  {((cell ^status |..| ^left <left> ^right <right> ^up <up>
  ^down <down> ^modified no) <cell>)
   (cell ^id <left> ^status |..| ^modified no)
   (cell ^id <right> ^status [*| ^modified no)
   (cell ^id <up> ^status [*| ^modified no)
   (cell ^id <down> ^status [*| ^modified no)
  -->
  (copy <cell> ^status [*| ^modified yes))

(p implement-kill-rule-all-neighbors
  (flag do-computation)
  (generation ^type desired ^value <x>)
  (generation ^type current ^value < <x>)
  {((cell ^status [*| ^left <left> ^right <right> ^up <up>

```

```

^down <down> ^modified no) <cell>
  (cell ^id <left> ^status [*] ^modified no)
  (cell ^id <right> ^status [*] ^modified no)
  (cell ^id <up> ^status [*] ^modified no)
  (cell ^id <down> ^status [*] ^modified no)
-->
  (copy <cell> ^status [*] ^modified yes))

(p set-modified-flags-1
  (flag set-modified-flags)
  ((cell ^modified nil) <cell>)
-->
  (modify <cell> ^modified no))

(p set-modified-flags-2
  (flag set-modified-flags)
  ((cell ^id <id> ^modified yes) <cell1>)
  ((cell ^id <id> ^modified no) <cell2>)
-->
  (modify <cell1> ^modified no)
  (remove <cell2>))

(p stage-computation-0
  (flag compute-cells)
-->
  (modify 1 set-modified-flags))

(p stage-computation-1
  (flag set-modified-flags)
-->
  (modify 1 do-computation))

(p stage-computation-2
  (flag do-computation)
-->
  (modify 1 print-cells))

```

E.2 Parallel version of the game of life

```

; Parallel version of the game life in PPL
; Each cell is represented as a tuple. It contains a field for its status
; and four fields for links to neighbors.
; The rules of life are:
; 1. if an organism has < 2 neighbors, it dies of loneliness
; 2. if it has 2 or 3 neighbors it lives
; 3. a new organism is born in a cell with three neighbors

```

```
; 4. if it has 4 neighbors, it dies of overcrowdedness.  
;  
; Version Notes: this is the most efficient parallel version  
; Only the tuples corresponding to cells whose value changes are modified.  
; The multiple-instantiation-firing semantics assumed allows all  
; updates to be made atomically and there is no need to copy tuples for the  
; purpose of atomicity.  
-----  
; Parallelized by: Anurag Acharya, acha@cs.cmu.edu  
-----  
-----  
; Tuple declarations  
-----  
; coordinates are needed for printing, id is needed for linking  
(literalize cell x-coordinate y-coordinate id status left right up down)  
  
; generation tuple is for counters and limits -- type is either current or  
; desired.  
(literalize generation type value)  
  
; flag tuple is used to control the sequencing -- values are print,  
; completed-print, compute-cells, completed  
(literalize flag value)  
  
; format tuple is used to store user format preferences -- values are  
; no and yes. yes prints the map every generation, no prints  
; the map only at the end of the simulation  
(literalize format value)  
  
; this tuple class is used for traversing the grid for printing purposes  
(literalize print-coordinates x-coordinate y-coordinate)  
-----  
; Productions  
-----  
  
(p start  
  (start)  
  -->  
  (make generation current 0)  
  (write "Generations to run simulations for. ")  
  (make generation desired (accept))  
  (write "Print after every generation ? ")  
  (make format (accept))  
  (make flag compute-cells))  
  
(p switch-to-next-generation
```

```

(generation ^type desired ^value <x>)
((generation ^type current ^value {<y> < <x>}) <cur-gen>)
((flag completed-print) <flag>)
-->
(modify <cur-gen> ^value (compute <y> + 1))
(modify <flag> compute-cells)

(p finished-simulation
(generation ^type desired ^value <x>)
(generation ^type current ^value <x>)
((flag completed-print) <flag>)
-->
(modify <flag> completed))

; of the next three productions, only one fires in a given cycle. If the
; "per-generation" production matches, it will be selected due to
; specificity, else the vanilla "switch-to" production will fire.
(p switch-to-print
(generation ^type desired ^value <x>)
(generation ^type current ^value <x>)
((flag compute-cells) <flag>)
-->
(modify <flag> print))

(p switch-to-print-final
(generation ^type desired ^value <x>)
(generation ^type current ^value <x>)
((flag compute-cells) <flag>)
-->
(modify <flag> print))

(p switch-to-print-per-generation
(generation ^type desired ^value <x>)
(generation ^type current ^value <x>)
((flag compute-cells) <flag>)
(format ^value no)
-->
(modify <flag> completed-print))

(p init-print
(flag print)
(generation ^type current ^value <x>)
-->
(make print-coordinates ^x-coordinate 0 ^y-coordinate 0)
(write generation <x> ":\n"))

(p print-next-cell-in-row
{print-coordinates ^x-coordinate <x> ^y-coordinate <y>} <coords>)

```

```

(cell ^x-coordinate <x> ^y-coordinate <y> ^status <status>)
-->
(write <status>)
(modify <coords> ^x-coordinate (compute <x> + 1)))

(p switch-rows
  { (print-coordinates ^x-coordinate <x> ^y-coordinate <y>) <coords>)
  -(cell ^x-coordinate <x>)
  -->
  (modify <coords> ^x-coordinate 0 ^y-coordinate (compute <y> + 1))
  (write (crlf)))

(p finalize-print
  { (flag print) <flag>)
  { (print-coordinates ^x-coordinate <x> ^y-coordinate <y>) <coords>)
  -(cell ^y-coordinate <y>)
  -->
  (modify <flag> completed-print)
  (remove <coords>)
  (write "\n\n"))

(pset kill-rule-0
  (parp implement-kill-rule-0
    (generation ^type desired ^value <x>)
    (generation ^type current ^value < <x>)
    (flag compute-cells)
    { (cell ^status [*] ^left <left> ^right <right> ^up <up>
    ^down <down>) <cell>)
    (cell ^id <left> ^status [.])
    (cell ^id <right> ^status [.])
    (cell ^id <up> ^status [.])
    (cell ^id <down> ^status [.])
    -->
    (modify <cell> ^status [.]))}

(pset kill-rule-left
  (parp implement-kill-rule-left
    (generation ^type desired ^value <x>)
    (generation ^type current ^value < <x>)
    (flag compute-cells)
    { (cell ^status [*] ^left <left> ^right <right> ^up <up>
    ^down <down>) <cell>)
    (cell ^id <left> ^status [*])
    (cell ^id <right> ^status [.])
    (cell ^id <up> ^status [.])
    (cell ^id <down> ^status [.])
    -->
    (modify <cell> ^status [.]))}

```

```
(pset kill-rule-right
  (parp implement-kill-rule-right
    (generation ^type desired ^value <xx>)
    (generation ^type current ^value < <xx>)
    (flag compute-cells)
    {((cell ^status '|`left <left> ^right <right> ^up <up>
      ^down <down>) <cell>)
     (cell ^id <left> ^status |.)
     (cell ^id <right> ^status '|`)
     (cell ^id <up> ^status |.)
     (cell ^id <down> ^status |.)}
    -->
    (modify <cell> ^status | |)))
(pset kill-rule-up
  (parp implement-kill-rule-up
    (generation ^type desired ^value <xx>)
    (generation ^type current ^value < <xx>)
    (flag compute-cells)
    {((cell ^status '|`left <left> ^right <right> ^up <up>
      ^down <down>) <cell>)
     (cell ^id <left> ^status | |)
     (cell ^id <right> ^status |.)
     (cell ^id <up> ^status '|`)
     (cell ^id <down> ^status |.)}
    -->
    (modify <cell> ^status |.|)))
(pset kill-rule-down
  (parp implement-kill-rule-down
    (generation ^type desired ^value <xx>)
    (generation ^type current ^value < <xx>)
    (flag compute-cells)
    {((cell ^status '|`left <left> ^right <right> ^up <up>
      ^down <down>) <cell>)
     (cell ^id <left> ^status |.)
     (cell ^id <right> ^status |.)
     (cell ^id <up> ^status |.)
     (cell ^id <down> ^status '|`)}
    -->
    (modify <cell> ^status |.|)))
(pset birth-rule-left-right-up
  (parp implement-birth-rule-left-right-up
    (generation ^type desired ^value <xx>)
    (generation ^type current ^value < <xx>)
    (flag compute-cells)
    {((cell ^status |.)`left <left> ^right <right> ^up <up>
      ^down <down>) <cell>})
```

```

  (cell ^id <left> ^status |*|)
  (cell ^id <right> ^status |*|)
  (cell ^id <up> ^status |*|)
  (cell ^id <down> ^status |.|)
  -->
  (modify <cell> ^status |*|))

(pset birth-rule-left-right-down
  (parp implement-birth-rule-left-right-down
    (generation ^type desired ^value <x>)
    (generation ^type current ^value < <x>)
    (flag compute-cells)
    ((cell ^status |.| ^left <left> ^right <right> ^up <up>
      ^down <down>) <cell>)
    (cell ^id <left> ^status |*|)
    (cell ^id <right> ^status |*|)
    (cell ^id <up> ^status |.|)
    (cell ^id <down> ^status |*|)
    -->
    (modify <cell> ^status |*|))

(pset birth-rule-left-up-down
  (parp implement-birth-rule-left-up-down
    (generation ^type desired ^value <x>)
    (generation ^type current ^value < <x>)
    (flag compute-cells)
    ((cell ^status |.| ^left <left> ^right <right> ^up <up>
      ^down <down>) <cell>)
    (cell ^id <left> ^status |*|)
    (cell ^id <right> ^status |.|)
    (cell ^id <up> ^status |*|)
    (cell ^id <down> ^status |*|)
    -->
    (modify <cell> ^status |*|))

(pset birth-rule-right-up-down
  (parp implement-birth-rule-right-up-down
    (generation ^type desired ^value <x>)
    (generation ^type current ^value < <x>)
    (flag compute-cells)
    ((cell ^status |.| ^left <left> ^right <right> ^up <up>
      ^down <down>) <cell>)
    (cell ^id <left> ^status |.|)
    (cell ^id <right> ^status |*|)
    (cell ^id <up> ^status |*|)
    (cell ^id <down> ^status |*|)
    -->
    (modify <cell> ^status |*|)))

```

```

{pset kill-rule-all-neighbors
  (parp implement-kill-rule-all-neighbors
    (generation ^type desired ^value <x>)
    (generation ^type current ^value <x>)
    (flag compute-cells)
    ((cell ^status [*] ^left <left> ^right <right> ^up <up>
      ^down <down> <cell>)
     (cell ^id <left> ^status [*])
     (cell ^id <right> ^status [*])
     (cell ^id <up> ^status [*])
     (cell ^id <down> ^status [*])
    -->
     (modify <cell> ^status [.]))}
```

E.3 Sequential version of the circuit simulator

; Sequential version of the gate level simulator.
 ; It incorporates a simplistic line delay model.
 ; Line delay is considered to be constant all over the ckt.
 ; It can handle the following devices:
 ; two input ands, two input nands, two input ors, two input nors,
 ; two input xors, nots
 ; It can be easily extended for all combinational devices.
 ; sequential ckt can be incorporated too but that would need
 ; some work
 ; It assumes that output of gates are not connected together
 ; in an implicit or.
 ;
 ; Version notes: To implement atomicity, it has to make copies of
 ; the lines that are used as outputs of the devices. It creates
 ; copies for only those lines whose value changes. At the end of the
 ; computation for a simulation cycle, the copies are merged back into
 ; the main circuit structure by removing all the old copies and modifying
 ; the flag on the new copies. The assumption of one device per output
 ; line can be easily removed but has not been removed for the following
 ; reasons:
 ; 1. it is satisfied by most real circuits. Circuits that don't
 ; satisfy it can be modelled by adding an or gate at the line.
 ; Since this simulator does not simulate timing behavior, the
 ; insertion of an extra OR gate does not make a difference.
 ; 2. Given the incremental nature of the match algorithms, the
 ; fewer tuples you modify, the better off you are. The output
 ; line is already being modified. Adding the modified/computed
 ; flag to the device would need more matching effort. This would
 ; increase parallelism but that would be fake parallelism as the

```
; additional computation is not really needed.  
;  
; Another optimization is to merge the new copies into the circuit  
; *before* simulating lines -- this avoids performing duplicate  
; computation for output lines.  
; Another optimization is to use a few more productions so that  
; variable tests can be replaced by constant tests. This again  
; reduces the match activity and reduces the amount of parallelism  
; available. But the computation avoided is not necessary and  
; therefore the parallelism eliminated is fake parallelism  
-----  
; Converted to PPL by: Anurag Acharya, acha@cs.cmu.edu  
; original version written by: Dan Neumann <dan@cs.umass.edu>  
-----  
;  
; Tuple declarations  
-----  
(literalize two-input-and-gate id input1 input2 output)  
(literalize two-input-or-gate id input1 input2 output)  
(literalize two-input-xor-gate id input1 input2 output)  
(literalize two-input-nor-gate id input1 input2 output)  
(literalize two-input-nand-gate id input1 input2 output)  
(literalize not-gate id input output)  
(literalize line id source sink modified)  
(literalize cycles-run value)  
(literalize cycles-desired value)  
(literalize flag value)  
(literalize results-desired value)  
-----  
; Productions  
-----  
(p startup  
  (start)  
  -->  
  (make cycles-run 0)  
  (write "cycles to run simulation for : ")  
  (make cycles-desired (accept))  
  (write "results to be printed: ")  
  (make results-desired (accept))  
  (make flag merge-lines-initial)  
  (remove 1))  
  
(p switch-from-initial-merge-lines  
  (flag merge-lines-initial)
```

```
-->
(modify 1 simulate-lines)

(p switch-from-line-to-devices
  (flag simulate-lines)
  -->
  (modify 1 simulate-devices))

(p switch-to-merge-new-lines
  (flag simulate-devices)
  -->
  (modify 1 merge-lines))

(p move-to-next-cycle
  (flag merge-lines)
  -(line ^modified yes)
  (cycles-desired <x>)
  (cycles-run (<y> <x>))
  -->
  (modify 3 (compute <y> + 1))
  (modify 1 simulate-lines))

(p completed-simulation-print-results
  (flag simulate-lines)
  (cycles-desired <x>)
  (cycles-run <x>)
  (results-desired yes)
  -->
  (write "Completed simulation of" <x> "cycles\n")
  (write "The final state is:" (crlf))
  (make flag print-results))

(p completed-simulation-without-results
  (flag simulate-lines)
  (cycles-desired <x>)
  (cycles-run <x>)
  -(results-desired yes)
  -->
  (write "Completed simulation of" <x> "cycles\n")
  (hs't))

(p merge-lines-initial
  (flag merge-lines-initial)
  ((line ^id <id> ^modified nil) <oldline>)
  -->
  (modify <oldline> ^modified no))

(p merge-lines
```

```

(flag merge-lines)
((line ^id <id> ^modified no) <oldline>)
((line ^id <id> ^modified yes) <newline>)
-->
(remove <oldline>)
(modify <newline> ^modified no)

(p simulate-line-transmission-turn-on
  (flag simulate-lines)
  (cycles-desired <x>)
  (cycles-run < <x>)
  ((line ^source 1 ^sink 0) <line>)
-->
(modify <line> ^sink 1))

(p simulate-line-transmission-turn-off
  (flag simulate-lines)
  (cycles-desired <x>)
  (cycles-run < <x>)
  ((line ^source 0 ^sink 1) <line>)
-->
(modify <line> ^sink 0))

(p simulate-two-input-and-gate-turn-on
; output = 0, input = 11
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-and-gate ^:input1 <input1> ^input2 <input2> ^output <output>)
  (line ^id <input1> ^sink 1 ^modified no)
  (line ^id <input2> ^sink 1 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
-->
(copy <output-line> ^source 1 ^modified yes))

(p simulate-two-input-and-gate-turn-off-1
; output = 1, input1 = 0
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-and-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 0 ^modified no)
  (line ^id <input2> ^sink 0 ^modified no)
  ((line ^id <output> ^source 1 ^modified no) <output-line>)
-->
(copy <output-line> ^source 0 ^modified yes))

```

```
(p simulate-two-input-and-gate-turn-off-2
  (flag simulate-devices)
  ; output = 1, input2 = 0
  (cycles-desired <xx>)
  (cycles-run < xx>)
  (two-input-and-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 0 ^modified no)
  (line ^id <input2> ^sink 1 ^modified no)
  ((line ^id <output> ^source 1 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 0 ^modified yes))

(p simulate-two-input-and-gate-turn-off-3
  (flag simulate-devices)
  ; output = 1, input2 = 0
  (cycles-desired <xx>)
  (cycles-run < xx>)
  (two-input-and-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 1 ^modified no)
  (line ^id <input2> ^sink 0 ^modified no)
  ((line ^id <output> ^source 1 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 0 ^modified yes))

(p simulate-two-input-or-gate-turn-off
  (flag simulate-devices)
  (cycles-desired <xx>)
  (cycles-run < xx>)
  (two-input-or-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 0 ^modified no)
  (line ^id <input2> ^sink 0 ^modified no)
  ((line ^id <output> ^source 1 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 0 ^modified yes))

(p simulate-two-input-or-gate-turn-on-1
  (flag simulate-devices)
  (cycles-desired <xx>)
  (cycles-run < xx>)
  (two-input-or-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 0 ^modified no)
  (line ^id <input2> ^sink 1 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
  -->
```

```
(copy <output-line> ^source 1 ^modified yes))

(p simulate-two-input-or-gate-turn-on-2
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-or-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  ((line ^id <input1> ^sink 1 ^modified no)
  ((line ^id <input2> ^sink 0 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 1 ^modified yes))

(p simulate-two-input-or-gate-turn-on-3
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-or-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  ((line ^id <input1> ^sink 1 ^modified no)
  ((line ^id <input2> ^sink 1 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 1 ^modified yes))

(p simulate-two-input-xor-gate-turn-off-1
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-xor-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  ((line ^id <input1> ^sink 0 ^modified no)
  ((line ^id <input2> ^sink 0 ^modified no)
  ((line ^id <output> ^source 1 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 0 ^modified yes))

(p simulate-two-input-xor-gate-turn-off-2
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-xor-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  ((line ^id <input1> ^sink 1 ^modified no)
  ((line ^id <input2> ^sink 1 ^modified no)
  ((line ^id <output> ^source 1 ^modified no) <output-line>)
  -->
```

```
(copy <output-line> ^source 0 ^modified yes))

(p simulate-two-input-xor-gate-turn-on-1
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-xor-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 0 ^modified no)
  (line ^id <input2> ^sink 1 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 1 ^modified yes))

(p simulate-two-input-xor-gate-turn-on-2
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-xor-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 1 ^modified no)
  (line ^id <input2> ^sink 0 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 1 ^modified yes))

(p simulate-two-input-nor-gate-turn-on
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-nor-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 0 ^modified no)
  (line ^id <input2> ^sink 0 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 1 ^modified yes))

(p simulate-two-input-nor-gate-turn-off-1
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-nor-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 1 ^modified no)
  (line ^id <input2> ^sink 0 ^modified no)
  ((line ^id <output> ^source 1 ^modified no) <output-line>)
  -->
```

```
(copy <output-line> ^source 0 ^modified yes))

(p simulate-two-input-nor-gate-turn-off-2
  (flag simulate-devices)
  (cycles-desired <xx>)
  (cycles-run < xx>)
  (two-input-nor-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 0 ^modified no)
  (line ^id <input2> ^sink 1 ^modified no)
  ((line ^id <output> ^source 1 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 0 ^modified yes))

(p simulate-two-input-nor-gate-turn-off-3
  (flag simulate-devices)
  (cycles-desired <xx>)
  (cycles-run < xx>)
  (two-input-nor-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 1 ^modified no)
  (line ^id <input2> ^sink 1 ^modified no)
  ((line ^id <output> ^source 1 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 0 ^modified yes))

(p simulate-two-input-nand-gate-turn-off
  (flag simulate-devices)
  (cycles-desired <xx>)
  (cycles-run < xx>)
  (two-input-nand-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 1 ^modified no)
  (line ^id <input2> ^sink 1 ^modified no)
  ((line ^id <output> ^source 1 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 0 ^modified yes))

(p simulate-two-input-nand-gate-turn-on-1
  (flag simulate-devices)
  (cycles-desired <xx>)
  (cycles-run < xx>)
  (two-input-nand-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 0 ^modified no)
  (line ^id <input2> ^sink 0 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
  -->
```

```

(copy <output-line> ^source 1 ^modified yes))

(p simulate-two-input-nand-gate-turn-on-2
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-nand-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 1 ^modified no)
  (line ^id <input2> ^sink 0 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 1 ^modified yes))

(p simulate-two-input-nand-gate-turn-on-3
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (two-input-nand-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink 0 ^modified no)
  (line ^id <input2> ^sink 1 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 1 ^modified yes))

(p simulate-not-gate-turn-off
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (not-gate ^input <input> ^output <output>)
  (line ^id <input> ^sink 1 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 0 ^modified yes))

(p simulate-not-gate-turn-on
  (flag simulate-devices)
  (cycles-desired <x>)
  (cycles-run < <x>)
  (not-gate ^input <input> ^output <output>)
  (line ^id <input> ^sink 0 ^modified no)
  ((line ^id <output> ^source 0 ^modified no) <output-line>)
  -->
  (copy <output-line> ^source 1 ^modified yes))

(p initiate-print
  (flag print-results)

```

```
-->
(modify 1 print-devices)

(p switch-to-print-lines
  (flag print-devices)
  -->
  (modify 1 print-lines))

(p finalize-print
  (flag print-lines)
  -->
  (halt))

(p print-and-gate-info
  (flag print-devices)
  (two-input-and-gate ^id <gate> ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink <value1>)
  (line ^id <input2> ^sink <value2>)
  (line ^id <output> ^source <value3>)
  -->
  (write "and gate" <gate> ": input1 = " <value1> "input2 = " <value2>
    "output = " <value3> (crlf)))

(p print-or-gate-info
  (flag print-devices)
  (two-input-or-gate ^id <gate> ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink <value1>)
  (line ^id <input2> ^sink <value2>)
  (line ^id <output> ^source <value3>)
  -->
  (write "or gate" <gate> ". input1 = " <value1> "input2 = " <value2>
    "output = " <value3> (crlf)))

(p print-nand-gate-info
  (flag print-devices)
  (two-input-nand-gate ^id <gate> ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink <value1>)
  (line ^id <input2> ^sink <value2>)
  (line ^id <output> ^source <value3>)
  -->
  (write "nand gate" <gate> ". input1 = " <value1> "input2 = " <value2>
    "output = " <value3> (crlf)))

(p print-xor-gate-info
  (flag print-devices)
```

```

(two-input-xor-gate ^id <gate> ^input1 <input1> ^input2 <input2>
  ^output <output>
  (line ^id <input1> ^sink <value1>)
  (line ^id <input2> ^sink <value2>)
  (line ^id <output> ^source <value3>)
  -->
  (write "xor gate" <gate> ": input1 = " <value1> "input2 = " <value2>
    "output = " <value3> (crlf))

(p print-nor-gate-info
  (flag print-devices)
  (two-input-nor-gate ^id <gate> ^input1 <input1> ^input2 <input2>
    ^output <output>
    (line ^id <input1> ^sink <value1>)
    (line ^id <input2> ^sink <value2>)
    (line ^id <output> ^source <value3>)
    -->
    (write "nor gate" <gate> ": input1 = " <value1> "input2 = " <value2>
      "output = " <value3> (crlf))

(p print-not-gate-info
  (flag print-devices)
  (not-gate ^id <gate> ^input <input> ^output <output>
  (line ^id <input> ^sink <value1>)
  (line ^id <output> ^source <value2>)
  -->
  (write "not gate" <gate> ": input = " <value1>
    "output = " <value2> (crlf))

(p print-line-info
  (flag print-lines)
  (line ^id <line> ^source <source> ^sink <sink>)
  -->
  (write "line" <line> ": source = " <source>
    "sink = " <sink> (crlf)))

```

E.4 Parallel version of the circuit simulator

```

; Parallel version of the gate level simulator
; It incorporates a simplistic line delay model
; Line delay is considered to be constant all over the ckt.
; It can handle the following devices:
; two input ands, two input nands, two input ors, two input nors,
; two input xors, nots
; It can be easily extended for all combinational devices.
; sequential ckts can be incorporated too but that would need

```

```
; some work
; It assumes that output of gates are not connected together
; in an implicit or.
;
; Version notes. this is the efficient parallel version of the program
; It modifies tuples for only those lines whose values change due to
; device operation or line transmission.
; The assumption of one device per output line can be easily removed but has
; not been removed for the following reasons:
; 1. it is satisfied by most real circuits. circuits that don't satisfy
;    it can be modelled by adding an or gate at the line. Since this
;    simulator does not simulate timing behavior, the insertion of the
;    extra OR gate does not make a difference.
;
; Another optimization is to use a few more productions so that variable
; tests can be replaced by constant tests. This again reduces the match
; activity and reduces the amount of parallelism available. But the computation
; avoided is not necessary and therefore the parallelism eliminated is fake
; parallelism.
;

-----  
; Parallelized by: Anurag Acharya, acha@cs.cmu.edu
-----  
  
-----  
; Tuple declarations
-----  
(literalize two-input-and-gate id input1 input2 output)
(literalize two-input-or-gate id input1 input2 output)
(literalize two-input-xor-gate id input1 input2 output)
(literalize two-input-nor-gate id input1 input2 output)
(literalize two-input-nand-gate id input1 input2 output)
(literalize not-gate id input output)
(literalize line id source sink)
(literalize cycles-run value)
(literalize cycles-desired value)
(literalize flag value)
(literalize results-desired value)

-----  
; Productions
-----  
(p startup
  (start)
  -->
  (make cycles-run 0)
  (write "cycles to run simulation for  ")
```

```
(make cycles-desired (accept))
(write "results to be printed : ")
(make results-desired (accept))
(make flag simulate-lines)
(remove 1))

(p move-to-next-cycle
  (cycles-desired <x>)
  (cycles-run (<y> <x>))
  (flag simulate-devices)
  -->
  (modify 2 (compute <y> + 1))
  (modify 3 simulate-lines))

(p completed-simulation-notice-1
  (cycles-desired <x>)
  (cycles-run <x>)
  (results-desired yes)
  -->
  (write "Completed simulation of" <x> "cycles\n")
  (write "The final state is:" (crlf))
  (make flag print-results))

(p completed-simulation-notice-2
  (cycles-desired <x>)
  (cycles-run <x>)
  (results-desired yes)
  -->
  (write "Completed simulation of" <x> "cycles\n")
  (halt))

(p switch-from-line-to-devices
  (flag simulate-lines)
  -->
  (modify 1 simulate-devices))

(pset print-results
  (parp print-and-gate-info
    (flag print-results)
    (two-input-and-gate ^id <gate> ^input1 <input1> ^input2 <input2>
                       ^output <output>)
    (line ^id <input1> ^sink <value1>)
    (line ^id <input2> ^sink <value2>)
    (line ^id <output> ^source <value3>)
    -->
    (write "and gate" <gate> ": input1 = " <value1> "input2 = " <value2>
          "output = " <value3> (crlf)))
```

```
(parp print-or-gate-info
  (flag print-results)
  (two-input-or-gate ^id <gate> ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink <value1>)
  (line ^id <input2> ^sink <value2>)
  (line ^id <output> ^source <value3>)
  -->
  (write "or gate" <gate> ": input1 = " <value1> "input2 = " <value2>
    "output = " <value3> (crlf))

(parp print-nand-gate-info
  (flag print-results)
  (two-input-nand-gate ^id <gate> ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink <value1>)
  (line ^id <input2> ^sink <value2>)
  (line ^id <output> ^source <value3>)
  -->
  (write "nand gate" <gate> ": input1 = " <value1> "input2 = " <value2>
    "output = " <value3> (crlf))

(parp print-xor-gate-info
  (flag print-results)
  (two-input-xor-gate ^id <gate> ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink <value1>)
  (line ^id <input2> ^sink <value2>)
  (line ^id <output> ^source <value3>)
  -->
  (write "xor gate" <gate> ". input1 = " <value1> "input2 = " <value2>
    "output = " <value3> (crlf))

(parp print-nor-gate-info
  (flag print-results)
  (two-input-nor-gate ^id <gate> ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (line ^id <input1> ^sink <value1>)
  (line ^id <input2> ^sink <value2>)
  (line ^id <output> ^source <value3>)
  -->
  (write "nor gate" <gate> ". input1 = " <value1> "input2 = " <value2>
    "output = " <value3> (crlf))

(parp print-not-gate-info
  (flag print-results)
  (not-gate ^id <gate> ^input <input> ^output <output>)
  (line ^id <input> ^sink <value1>)
```

```
(line ^id <output> ^source <value2>)
-->
(write "not gate" <gate> ": input = " <value1>
      "output = " <value2> (crlf))

(parp print-line-info
  (flag print-results)
  (line ^id <line> ^source <source> ^sink <sink>)
  -->
  (write "line" <line> ": source = " <source>
        "sink = " <sink> (crlf))

(p completed-simulation-notice-3
  (flag print-results)
  -->
  (halt))

{pset two-input-and-gate-turn-on

(parp simulate-two-input-and-gate-turn-on
  (cycles-desired <x>)
  (two-input-and-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (cycles-run < <x>)
  (line ^id <input1> ^sink 1)
  (line ^id <input2> ^sink 1)
  ((line ^id <output> ^source 0) <output-line>)
  (flag simulate-devices)
  -->
  (modify <output-line> ^source 1))

(pset two-input-and-gate-turn-off-1
(parp simulate-two-input-and-gate-turn-off-1
  (cycles-desired <x>)
  (two-input-and-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (cycles-run < <x>)
  (line ^id <input1> ^sink 0)
  (line ^id <input2> ^sink 0)
  ((line ^id <output> ^source 1) <output-line>)
  (flag simulate-devices)
  -->
  (modify <output-line> ^source 0))

(pset two-input-and-gate-turn-off-2
(parp simulate-two-input-and-gate-turn-off-2
  (cycles-desired <x>)
```

```

(two-input-and-gate ^input1 <input1> ^input2 <input2>
                     ^output <output>)
(cycles-run < <x>)
  (line ^id <input1> ^sink 0)
  (line ^id <input2> ^sink 1)
  ((line ^id <output> ^source 1) <output-line>)
  (flag simulate-devices)
-->
(modify <output-line> ^source 0))

(pset two-input-and-gate-turn-off-3
(parp simulate-two-input-and-gate-turn-off-3
  (cycles-desired <x>)
  (two-input-and-gate ^input1 <input1> ^input2 <input2>
                     ^output <output>)
  (cycles-run < <x>)
  (line ^id <input1> ^sink 1)
  (line ^id <input2> ^sink 0)
  ((line ^id <output> ^source 1) <output-line>)
  (flag simulate-devices)
-->
(modify <output-line> ^source 0))

(pset two-input-or-gate-turn-off
(parp simulate-two-input-or-gate-turn-off
  (cycles-desired <x>)
  (two-input-or-gate ^input1 <input1> ^input2 <input2>
                     ^output <output>)
  (cycles-run < <x>)
  (line ^id <input1> ^sink 0)
  (line ^id <input2> ^sink 0)
  ((line ^id <output> ^source 1) <output-line>)
  (flag simulate-devices)
-->
(modify <output-line> ^source 0))

(pset two-input-or-gate-turn-on-1
(parp simulate-two-input-or-gate-turn-off-1
  (cycles-desired <x>)
  (two-input-or-gate ^input1 <input1> ^input2 <input2>
                     ^output <output>)
  (cycles-run < <x>)
  (line ^id <input1> ^sink 1)
  (line ^id <input2> ^sink 0)
  ((line ^id <output> ^source 0) <output-line>)
  (flag simulate-devices)
-->
(modify <output-line> ^source 1)))

```

```
(pset two-input-or-gate-turn-on-2
  (parp simulate-two-input-or-gate-turn-on-2
    (cycles-desired <x>)
    (two-input-or-gate ^input1 <input1> ^input2 <input2>
                      ^output <output>)
    (cycles-run < <x>)
    (line ^id <input1> ^sink 0)
    (line ^id <input2> ^sink 1)
    ((line ^id <output> ^source 0) <output-line>)
    (flag simulate-devices)
    -->
    (modify <output-line> ^source 1)))

(pset two-input-or-gate-turn-on-3
  (parp simulate-two-input-or-gate-turn-on-3
    (cycles-desired <x>)
    (two-input-or-gate ^input1 <input1> ^input2 <input2>
                      ^output <output>)
    (cycles-run < <x>)
    (line ^id <input1> ^sink 1)
    (line ^id <input2> ^sink 1)
    ((line ^id <output> ^source 0) <output-line>)
    (flag simulate-devices)
    -->
    (modify <output-line> ^source 1)))

(pset two-input-xor-gate-simulator-turn-off-1
  (parp simulate-two-input-xor-gate-turn-off-1
    (cycles-desired <x>)
    (two-input-xor-gate ^input1 <input1> ^input2 <input2>
                      ^output <output>)
    (cycles-run < <x>)
    (line ^id <input1> ^sink 0)
    (line ^id <input2> ^sink 0)
    ((line ^id <output> ^source 1) <output-line>)
    (flag simulate-devices)
    -->
    (modify <output-line> ^source 0)))

(pset two-input-xor-gate-simulator-turn-off-2
  (parp simulate-two-input-xor-gate-turn-off-2
    (cycles-desired <x>)
    (two-input-xor-gate ^input1 <input1> ^input2 <input2>
                      ^output <output>)
    (cycles-run < <x>)
    (line ^id <input1> ^sink 1)
    (line ^id <input2> ^sink 1)
```

```
((line ^id <output> ^source 1) <output-line>)
(iflag simulate-devices)
-->
(modify <output-line> ^source 0))

(pset two-input-xor-gate-simulator-turn-on-1
(parps simulate-two-input-xor-gate-turn-on-1
(cycles-desired <x>)
(two-input-xor-gate ^input1 <input1> ^input2 <input2>
^output <output>)
(cycles-run < <x>)
(line ^id <input1> ^sink 0)
(line ^id <input2> ^sink 1)
((line ^id <output> ^source 0) <output-line>)
(iflag simulate-devices)
-->
(modify <output-line> ^source 1))

(pset two-input-xor-gate-simulator-turn-on-2
(parps simulate-two-input-xor-gate-turn-on-2
(cycles-desired <x>)
(two-input-xor-gate ^input1 <input1> ^input2 <input2>
^output <output>)
(cycles-run < <x>)
(line ^id <input1> ^sink 1)
(line ^id <input2> ^sink 0)
((line ^id <output> ^source 0) <output-line>)
(iflag simulate-devices)
-->
(modify <output-line> ^source 1))

(pset two-input-nor-gate-simulator-turn-on
(parps simulate-two-input-nor-gate-turn-on
(cycles-desired <x>)
(two-input-nor-gate ^input1 <input1> ^input2 <input2>
^output <output>)
(cycles-run < <x>)
(line ^id <input1> ^sink 0)
(line ^id <input2> ^sink 0)
((line ^id <output> ^source 0) <output-line>)
(iflag simulate-devices)
-->
(modify <output-line> ^source 1))

(pset two-input-nor-gate-simulator-turn-off-1
(parps simulate-two-input-nor-gate-turn-off-1
(cycles-desired <x>)
(two-input-nor-gate ^input1 <input1> ^input2 <input2>
```

```
        ^output <output>)
(cycles-run < <x>)
(line ^id <input1> ^sink 0)
(line ^id <input2> ^sink 1)
((line ^id <output> ^source 1) <output-line>)
(flag simulate-devices)
-->
(modify <output-line> ^source 0))

(pset two-input-nor-gate-simulator-turn-off-2
(parp simulate-two-input-nor-gate-turn-off-2
  (cycles-desired <x>)
  (two-input-nor-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (cycles-run < <x>)
  (line ^id <input1> ^sink 1)
  (line ^id <input2> ^sink 0)
  ((line ^id <output> ^source 1) <output-line>)
  (flag simulate-devices)
-->
(modify <output-line> ^source 0))

(pset two-input-nor-gate-simulator-turn-off-3
(parp simulate-two-input-nor-gate-turn-off-3
  (cycles-desired <x>)
  (two-input-nor-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (cycles-run < <x>)
  (line ^id <input1> ^sink 1)
  (line ^id <input2> ^sink 1)
  ((line ^id <output> ^source 1) <output-line>)
  (flag simulate-devices)
-->
(modify <output-line> ^source 0))

(pset two-input-nand-gate-simulator-turn-on-1
(parp simulate-two-input-nand-gate-turn-on-1
  (cycles-desired <x>)
  (two-input-nand-gate ^input1 <input1> ^input2 <input2>
    ^output <output>)
  (cycles-run < <x>)
  (line ^id <input1> ^sink 0)
  (line ^id <input2> ^sink 0)
  ((line ^id <output> ^source 0) <output-line>)
  (flag simulate-devices)
-->
(modify <output-line> ^source 1)))
```

```
{pset two-input-nand-gate-simulator-turn-on-2
  (parp simulate-two-input-nand-gate-turn-on-2
    (cycles-desired <xx>)
    (two-input-nand-gate ^input1 <input1> ^input2 <input2>
      ^output <output>)
    (cycles-run < <xx>)
    (line ^id <input1> ^sink 0)
    (line ^id <input2> ^sink 1)
    ((line ^id <output> ^source 0) <output-line>)
    (flag simulate-devices)
    -->
    (modify <output-line> ^source 1)))

{pset two-input-nand-gate-simulator-turn-on-3
  (parp simulate-two-input-nand-gate-turn-on-3
    (cycles-desired <xx>)
    (two-input-nand-gate ^input1 <input1> ^input2 <input2>
      ^output <output>)
    (cycles-run < <xx>)
    (line ^id <input1> ^sink 1)
    (line ^id <input2> ^sink 1)
    ((line ^id <output> ^source 0) <output-line>)
    (flag simulate-devices)
    -->
    (modify <output-line> ^source 1)))

{pset two-input-nand-gate-simulator-turn-off
  (parp simulate-two-input-nand-gate-turn-off
    (cycles-desired <xx>)
    (two-input-nand-gate ^input1 <input1> ^input2 <input2>
      ^output <output>)
    (cycles-run < <xx>)
    (line ^id <input1> ^sink 1)
    (line ^id <input2> ^sink 1)
    ((line ^id <output> ^source 1) <output-line>)
    (flag simulate-devices)
    -->
    (modify <output-line> ^source 0)))

{pset not-gate-simulator-turn-on
  (parp simulate-not-gate-turn-on
    (cycles-desired <xx>)
    (not-gate ^input <input> ^output <output>)
    (cycles-run < <xx>)
    (line ^id <input> ^sink 0)
    ((line ^id <output> ^source 0) <output-line>)
    (flag simulate-devices)
    -->
```

```

(modify <output-line> ^source 1))

(pset not-gate-simulator-turn-off
(parp simulate-not-gate-turn-off
  (cycles-desired <>)
  (not-gate ^input <input> ^output <output>)
  (cycles-run < <>)
  (line ^id <input> ^sink 1)
  ((line ^id <output> ^source 1) <output-line>)
  (flag simulate-devices)
-->
(modify <output-line> ^source 0))

(pset line-simulator-turn-on
(parp simulate-line-transmission-turn-on
  (cycles-desired <x>)
  (cycles-run < <x>)
  ((line ^source 1 ^sink 0) <line>)
  (flag simulate-lines)
-->
(modify <line> ^sink 1))

(pset line-simulator-turn-off
(parp simulate-line-transmission-turn-off
  (cycles-desired <x>)
  (cycles-run < <x>)
  ((line ^source 0 ^sink 1) <line>)
  (flag simulate-lines)
-->
(modify <line> ^sink 0)))

```

E.5 Sequential version of waltz

```

; Program to interpret a line drawing. It takes the junction labelling
; as input and generates a consistent line labelling.
;-----
; Original program written by: Toru Ishida
;
; Modified by Dan Neiman, COINS Dept., University of Massachusetts
; 11/16/90: Added possible-line-label element. One element is added for
; each possible labelling of each end of each line. This allows easy
; testing for consistent line labelling without proliferation of rules.
; There are now only five rules in the reduce phase which correspond
; more closely to the actual knowledge being applied.
;
; Modified by: Anurag Acharya acha@cs.cmu.edu

```

```

;
; 1. moved the data to the file to be loaded -- this allows me to scale the
;    input
; 2. Reordered the condition elements for the enumerate-possible-candidates
;    and consistent-* productions. Lifted the staging condition to the
;    top -- this is needed for correctness. Else, loop termination detection
;    v;    does not work and declares termination before any iterations have been
;    executed.
; 3. added printing production -- in particular order. That is necessary
;    for some measure of realism. Everything can't be parallel in real life
;    and any labelling program will have to read the labels in some way.
;-----

;-----  

; Tuple declarations  

;-----  

(literalize possible-junction-label junction-type label-id line-1 line-2  

    line-3)  

(literalize junction junction-type junction-ID line-ID-1 line-ID-2 line-ID-3)  

(literalize labelling-candidate junction-ID line-1 line-2 line-3 1-c-ID)  

(literalize possible-line-label line candidate junction label)

(p start-Waltz
  (start)
  -->
  (make stage enumerate-possible-candidates))

(p enumerate-possible-candidates
  (stage enumerate-possible-candidates)
  (possible-junction-label ^junction-type <)-type>
    ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
  (junction ^junction-type <j-type> ^junction-ID <j-ID>
    ^line-ID-1 <l1> ^line-ID-2 <l2> ^line-ID-3 <l3>)
  -(labelling-candidate ^junction-ID <j-ID>
    ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
  -->
  (bind <1-c-ID> (genatom))
  (make labelling-candidate ^junction-ID <j-ID> ^1-c-ID <1-c-ID>
    ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
  (make possible-line-label ^line <l1> ^label <line-1> ^candidate <1-c-ID>
    ^junction <j-ID>)
  (make possible-line-label ^line <l2> ^label <line-2> ^candidate <1-c-ID>
    ^junction <j-ID>)
  (make possible-line-label ^line <l3> ^label <line-3> ^candidate <1-c-ID>
    ^junction <j-ID>))

(p go-to-reduce-candidates

```

```

        (stage enumerate-possible-candidates)
        -->
        (remove 1)
        (make stage reduce-candidates))

; the following productions apply the consistent labelling constraints
(p consistent-plus
  (stage reduce-candidates)
  { (possible-line-label ^line <line> ^junction <junction>
      ^label + ^candidate <c>) <line>)
    { (labelling-candidate ^l-c-ID <c>) <l-c>)
      -(possible-line-label ^line <line> ^junction <> <junction> ^label +)
      -->
      (remove <line>)
      (remove <l-c>))

  (p consistent-minus
    (stage reduce-candidates)
    { (possible-line-label ^line <line> ^junction <junction>
        ^label - ^candidate <c>) <line>)
      { (labelling-candidate ^l-c-ID <c>) <l-c>)
        -(possible-line-label ^line <line> ^junction <> <junction> ^label -)
        -->
        (remove <line>)
        (remove <l-c>))

    (p consistent-in-out
      (stage reduce-candidates)
      { (possible-line-label ^line <line> ^junction <junction>
          ^label in ^candidate <c>) <line>)
        { (labelling-candidate ^l-c-ID <c>) <l-c>)
          -(possible-line-label ^line <line> ^junction <> <junction> ^label out)
          -->
          (remove <line>)
          (remove <l-c>))

    (p consistent-out-in
      (stage reduce-candidates)
      { (possible-line-label ^line <line> ^junction <junction>
          ^label out ^candidate <c>) <line>)
        { (labelling-candidate ^l-c-ID <c>) <l-c>)
          -(possible-line-label ^line <line> ^junction <> <junction> ^label in)
          -->
          (remove <line>)
          (remove <l-c>))

; When a labelling-candidate is deleted, we want to also delete all possible
; line labels associated with that labelling-candidate

```

```

(p eliminate-line-labels
  (stage reduce-candidates)
  ((possible-line-label ^candidate <c>) <old>)
  -(labelling-candidate ^l-c-ID <c>)
  -->
  (remove <old>))

(p go-to-print-out
  (stage reduce-candidates)
  -->
  (remove 1)
  (make stage print-out))

(p print-out
  (stage print-out)
  (possible-line-label ^junction <junction> ^line <line> ^label <label>)
  -->
  (write "junction " <junction> <line> <label> (crlf)))

; fires last by specificity
(p halt
  (stage print-out)
  -->
  (halt))

```

E.6 Parallel version of waltz

```

; Program to interpret a line drawing. It takes the junction labelling
; as input and generates a consistent line labelling.
;-----
; Original program written by: Toru Ishida
;
; Modified by Dan Neiman, COINS Dept., University of Massachusetts
; 11/16/90: Added possible-line-label element. One element is added for
; each possible labelling of each end of each line. This allows easy
; testing for consistent line labelling without proliferation of rules.
; There are now only five rules in the reduce phase which correspond
; more closely to the actual knowledge being applied.
;
; Parallelized by: Anurag Acharya acha@cs.cmu.edu
; 1. moved the data to the file to be loaded -- this allows me to scale the
;    input
; 2. Reordered the condition elements for the enumerate-possible-candidates
;    and consistent-* productions. Lifted the staging condition to the
;    top -- this is needed for correctness. Else, loop termination detection
;    does not work and declares termination before any iterations have been

```

```

; executed.
; 3. added printing production -- in particular order. That is necessary
; for some measure of realism. Everything can't be parallel in real life
; and any labelling program will have to read the labels in some way.
;-----

;-----  

; Tuple declarations  

;-----  

(literalize possible-junction-label junction-type label-id line-1 line-2  

    line-3)  

(literalize junction junction-type junction-ID line-ID-1 line-ID-2 line-ID-3)  

(literalize labelling-candidate junction-ID line-1 line-2 line-3 l-c-ID)  

(literalize possible-line-label line candidate junction label)

(p start-Waltz
  (start)
  -->
  (make stage enumerate-possible-candidates))

(p go-to-reduce-candidates
  (stage enumerate-possible-candidates)
  -->
  (remove 1)
  (make stage reduce-candidates))

(parpa consistent-plus
  (stage reduce-candidates)
  ((possible-line-label ^line <line> ^junction <junction>
    ^label + ^candidate <c> <line>)
   (labelling-candidate ^l-c-ID <c>) <l-c>)
  -(possible-line-label ^line <line> ^junction <> <junction> ^label +)
  -->
  (remove <line>)
  (remove <l-c>))

(parpa consistent-minus
  (stage reduce-candidates)
  ((possible-line-label ^line <line> ^junction <junction>
    ^label - ^candidate <c> <line>)
   (labelling-candidate ^l-c-ID <c>) <l-c>)
  -(possible-line-label ^line <line> ^junction <> <junction> ^label -)
  -->
  (remove <line>)
  (remove <l-c>))

(parpa consistent-in-out
  (stage reduce-candidates))

```

```
((possible-line-label ^line <line> ^junction <junction>
  ^label in ^candidate <c>) <line>)
((labelling-candidate ^l-c-ID <c>) <l-c>)
-(possible-line-label ^line <line> ^junction <> <junction> ^label out)
-->
(remove <line>)
(remove <l-c>))

(parp consistent-out-in
  (stage reduce-candidates)
  ((possible-line-label ^line <line> ^junction <junction>
    ^label out ^candidate <c>) <line>)
  ((labelling-candidate ^l-c-ID <c>) <l-c>)
  -(possible-line-label ^line <line> ^junction <> <junction> ^label in)
  -->
  (remove <line> )
  (remove <l-c>))

(parp eliminate-line-labels
  (stage reduce-candidates)
  ((possible-line-label ^candidate <c>) <old>)
  -(labelling-candidate ^l-c-ID <c>)
  -->
  (remove <old>))

(p go-to-print-out
  (stage reduce-candidates)
  -->
  (remove 1)
  (make stage print-out))

(parp print-out
  (stage print-out)
  (possible-line-label ^junction <junction> ^line <line> ^label <label>)
  -->
  (write "junction " <junction> <line> <label> (crlf)))

(p halt
  (stage print-out)
  -->
  (halt))

(pset enumerate
  (parp enumerate-possible-candidates
    (stage enumerate-possible-candidates)
    (possible-junction-label ^junction-type <j-type>
      ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
    (junction ^junction-type <j-type> ^junction-ID <j-ID>)
```

```

    ^line-ID-1 <l1> ^line-ID-2 <l2> ^line-ID-3 <l3>)
  -(labelling-candidate ^junction-ID <j-ID>
    ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
  -->
  (bind <l-c-ID> (genatom))
  (make labelling-candidate ^junction-ID <j-ID> ^l-c-ID <l-c-ID>
    ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
  (make possible-line-label ^line <l1> ^label <line-1> ^candidate <l-c-ID>
    ^junction <j-ID>)
  (make possible-line-label ^line <l2> ^label <line-2> ^candidate <l-c-ID>
    ^junction <j-ID>)
  (make possible-line-label ^line <l3> ^label <line-3> ^candidate <l-c-ID>
    ^junction <j-ID>)))

```

E.7 Sequential version of hotel

```

; Program to model the operation of a hotel.
;
; Original version by: Steve Kuo, University of Southern California
;
; Converted to PPL by: Anurag Acharya, acha@cs.cmu.edu
;-----
; The original version had a fixed number of floors and had applied
; copy and constraint very extensively. So much so that C compilers
; refused to compile files that large. There are several other
; optimizations including splitting room-linen and room-furniture
; tuples so that the affect set size goes down.
;-----

;-----
; Tuple declarations
;-----
(literalize customer name status reservation arrival-date
  departure-date num-person room-num cash)
(literalize room room-num status phase departure-date room-type
  floor name)
(literalize hallway floor restroom section)
(literalize rate room-type num-person rate)
(literalize reservation status name num-person
  arrival-date departure-date)
(literalize room-towel room-num number)
(literalize room-sheet room-num number)
(literalize room-pillow-case room-num number)
(literalize room-trash-can room-num status)
(literalize room-blanket room-num status)
(literalize room-bedsheet room-num status)

```

```
(literalize room-bathroom room-num status)
(literalize room-dresser room-num status)
(literalize room-table room-num status)
(literalize room-chairs room-num status)
(literalize room-vacuum room-num status)
(literalize dirty-sheet floor quantity)
(literalize washed-sheet floor quantity)
(literalize clean-sheet floor quantity)
(literalize dirty-towel floor quantity)
(literalize washed-towel floor quantity)
(literalize clean-towel floor quantity)
(literalize dirty-pillow-case floor quantity)
(literalize washed-pillow-case floor quantity)
(literalize clean-pillow-case floor quantity)
(literalize soup name quantity)
(literalize dish name quantity)
(literalize veg name quantity)
(literalize order-chowder conference quantity)
(literalize order-chicken conference quantity)
(literalize order-hamburger conference quantity)
(literalize order-steak conference quantity)
(literalize order-seafood conference quantity)
(literalize order-salad conference quantity)
(literalize order-fruit conference quantity)
(literalize conference-chowder conference quantity)
(literalize conference-chicken conference quantity)
(literalize conference-hamburger conference quantity)
(literalize conference-steak conference quantity)
(literalize conference-seafood conference quantity)
(literalize conference-salad conference quantity)
(literalize conference-fruit conference quantity)
(literalize table conference number clean)
(literalize table-cloth table-number number)
(literalize table-plate table-number number)
(literalize table-knife table-number number)
(literalize table-fork table-number number)
(literalize table-napkin table-number number)
(literalize table-chairs table-number number)
(literalize chair number)
(literalize cloth number)
(literalize plate number)
(literalize fork number)
(literalize knife number)
(literalize napkin number)
(literalize dish-counter-hamburger conference number quantity)
(literalize dish-counter-steak conference number quantity)
(literalize dish-counter-seafood conference number quantity)
(literalize salad-ba conference number quantity)
```

```
(literalize fruit-bar conference number quantity)
(literalize soup-bar conference name status)
(literalize coffee-machine conference number power coffee
    beans pot water-level)
(literalize context name)
(literalize date day)

(p 0.initialize
  (start)
  -->
  (make context ^name clean-hallway)
  (make context ^name conference)
  (make context ^name check-out))

(p 1.check-out
  (context ^name check-out)
  (date ^day <d>)
  ((customer ^status staying ^arrival-date <ad>
^departure-date <d> ^room-num <r> ^cash <c>) <customer>)
  ((room ^room-num <r> ^room-type <rt> ^status occupied) <room>)
  (rate ^room-type <rt> ^rate <rate>)
  ((room-towel ^room-num <r>) <towel>)
  ((room-sheet ^room-num <r>) <sheet>)
  ((room-pillow-case ^room-num <r>) <pillow-case>)
  ((room-trash-can ^room-num <r>) <trash-can>)
  ((room-blanket ^room-num <r>) <blanket>)
  ((room-bedspread ^room-num <r>) <bed-spread>)
  ((room-bathroom ^room-num <r>) <bathroom>)
  ((room-dresser ^room-num <r>) <dresser>)
  ((room-table ^room-num <r>) <table>)
  ((room-chairs ^room-num <r>) <chairs>)
  ((room-vacuum ^room-num <r>) <vacuum>)
-->
  (modify <customer> ^status check-out
    ^cash (compute <c> - (<rate> * (<d> - <ad>))))
  (modify <room> ^status change ^phase maid-laundry)
  (modify <towel> ^number 8)
  (modify <sheet> ^number 4)
  (modify <pillow-case> ^number 4)
  (modify <trash-can> ^status not-empty)
  (modify <blanket> ^status not-made)
  (modify <bed-spread> ^status not-made)
  (modify <bathroom> ^status dirty)
  (modify <dresser> ^status dirty)
  (modify <table> ^status dirty)
  (modify <chairs> ^status dirty)
  (modify <vacuum> ^status dirty))
```

```

(p 19.staying-over
  (context ^name check-out)
  (date ^day <d>)
  ((customer ^status staying ^room-num <r>
    ^departure-date <dd> <d>) <customer>)
  ((room ^room-num <r> ^status occupied) <room>)
  ((room-towel ^room-num <r>) <towel>)
  ((room-sheet ^room-num <r>) <sheet>)
  ((room-pillow-case ^room-num <r>) <pillow-case>)
  ((room-trash-can ^room-num <r>) <trash-can>)
  ((room-blanket ^room-num <r>) <blanket>)
  ((room-bedspread ^room-num <r>) <bed-spread>)
  ((room-bathroom ^room-num <r>) <bathroom>)
  ((room-dresser ^room-num <r>) <dresser>)
  ((room-table ^room-num <r>) <table>)
  ((room-chairs ^room-num <r>) <chairs>)
  ((room-vacuum ^room-num <r>) <vacuum>)
-->
  (modify <room> ^status make-up ^phase maid-cleaning)
  (modify <towel> ^number 0)
  (modify <sheet> ^number 0)
  (modify <pillow-case> ^number 0)
  (modify <trash-can> ^status not-empty)
  (modify <blanket> ^status not-made)
  (modify <bed-spread> ^status not-made)
  (modify <bathroom> ^status dirty)
  (modify <dresser> ^status dirty)
  (modify <table> ^status dirty)
  (modify <chairs> ^status dirty)
  (modify <vacuum> ^status dirty))

(p 73.done-check-out
  ((context ^name check-out) <context>)
  -(room ^status occupied)
-->
  (modify <context> ^name maid-laundry))

(p 74.strip-towel
  (context ^name maid-laundry)
  (room ^room-num <r> ^phase maid-laundry ^floor <floor>)
  ((room-towel ^room-num <r> ^number (<t> > 0)) <room-linen>)
  ((dirty-towel ^floor <floor> ^quantity <q>) <towel>)
-->
  (modify <room-linen> ^number 0)
  (modify <towel> ^quantity (compute <q> + <t>)))

(p 75.strip-sheet
  (context ^name maid-laundry)

```

```

(room ^room-num <r> ^phase maid-laundry ^floor <floor>
  ((room-sheet ^room-num <r> ^number (<> > 0)) <room-linen>
   ((dirty-sheet ^floor <floor> ^quantity <q>) <sheet>)
-->
  (modify <room-linen> ^number 0)
  (modify <sheet> ^quantity (compute <q> + <s>)))

(p 76.strip-pillow-case
  (context ^name maid-laundry)
  (room ^room-num <r> ^phase maid-laundry ^floor <floor>
    ((room-pillow-case ^room-num <r> ^number (<p> > 0)) <room-linen>
     ((dirty-pillow-case ^floor <floor> ^quantity <q>) <case>)
-->
  (modify <room-linen> ^number 0)
  (modify <case> ^quantity (compute <q> + <p>)))

(p 77.finish-laundry-room
  (context ^name maid-laundry)
  ((room ^room-num <r> ^phase maid-laundry) <room>
   (room-towel ^room-num <r> ^number 0)
   (room-sheet ^room-num <r> ^number 0)
   (room-pillow-case ^room-num <r> ^number 0)
-->
  (modify <room> ^phase maid-cleaning))

(p 146 done-maid-laundry
  ((context ^name maid-laundry) <context>
   -(room ^phase maid-laundry)
-->
  (make context ^name laundry)
  (modify <context> ^name maid-cleaning))

(p 147.towel-needed
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  ((room-towel ^room-num <r> ^number < 8) <room-linen>)
-->
  (modify <room-linen> ^number 8))

(p 148.make-sheet
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  ((room-sheet ^room-num <r> ^number < 4) <room-linen>)
-->
  (modify <room-linen> ^number 4))

(p 149 make-pillow
  (context ^name maid-cleaning)

```

```
(room ^room-num <r> ^phase maid-cleaning)
((room-pillow-case ^room-num <r> ^number < 4) <room-linen>)
-->
(modify <room-linen> ^number 4))

(p 150.clean-trash-can
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  ((room-trash-can ^room-num <r> ^status not-empty) <room-linen>)
-->
(modify <room-linen> ^status empty))

(p 151.make-blanket
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  ((room-blanket ^room-num <r> ^status not-made) <room-linen>)
-->
(modify <room-linen> ^status made))

(p 152.make-bed-spread
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  ((room-bedsheet ^room-num <r> ^status not-made) <room-linen>)
-->
(modify <room-linen> ^status made))

(p 153.clean-bathroom
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  ((room-bathroom ^room-num <r> ^status dirty) <room-furniture>)
-->
(modify <room-furniture> ^status clean))

(p 154.clean-dresser
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  ((room-dresser ^room-num <r> ^status dirty) <room-furniture>)
-->
(modify <room-furniture> ^status clean))

(p 155.clean-table
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  ((room-table ^room-num <r> ^status dirty) <room-furniture>)
-->
(modify <room-furniture> ^status clean))

(p 156.clean-chairs
```

```

(context ^name maid-cleaning)
(=room ^room-num <r> ^phase maid-cleaning)
((room-chairs ^room-num <r> ^status dirty) <room-furniture>)
-->
(modify <room-furniture> ^status clean))

(p 157.vacuum-3
(context ^name maid-cleaning)
(room ^room-num <r> ^phase maid-cleaning)
((room-vacuum ^room-num <r> ^status dirty) <room-furniture>)
-->
(modify <room-furniture> ^status done))

(p 158.finish-room-changing
(context ^name maid-cleaning)
((room ^room-num <r> ^status change
    ^phase maid-cleaning) <room>)
(room-towel ^room-num <> ^number 8)
(room-sheet ^room-num <r> ^number 4)
(room-pillow-case ^room-num <r> ^number 4)
(room-trash-can ^room-num <r> ^status empty)
(room-blanket ^room-num <r> ^status made)
(room-bedsheet ^room-num <r> ^status made)
(room-bathroom ^room-num <r> ^status clean)
(room-dresser ^room-num <r> ^status clean)
(room-table ^room-num <r> ^status clean)
(room-chairs ^room-num <r> ^status clean)
(room-vacuum ^room-num <r> ^status done)
-->
(modify <room> ^status vacant ^phase clean))

(p 159.finish-room-make-up
(context ^name maid-cleaning)
((room ^room-num <r> ^status make-up
    ^phase maid-cleaning) <room>)
(room-towel ^room-num <r> ^number 8)
(room-sheet ^room-num <r> ^number 4)
(room-pillow-case ^room-num <r> ^number 4)
(room-trash-can ^room-num <r> ^status empty)
(room-blanket ^room-num <r> ^status made)
(room-bedsheet ^room-num <r> ^status made)
(room-bathroom ^room-num <r> ^status clean)
(room-dresser ^room-num <r> ^status clean)
(room-table ^room-num <r> ^status clean)
(room-chairs ^room-num <r> ^status clean)
(room-vacuum ^room-num <r> ^status done)
-->
(modify <room> ^status occupied ^phase clean))

```

```
(p 381.done-maid-cleaning
  ((context ^name maid-cleaning) <context>,
  -(room ^phase maid-cleaning)
-->
  (make context ^name reservation)
  (modify <context> ^name done-maid-cleaning))

(p 382.clean-restroom
  ((context ^name clean-hallway)
  ((hallway ^restroom {<n> > 0}) <hallway>)
-->
  (modify <hallway> ^restroom (compute <n> - 1)))

(p 400.done-clean-hallway
  ((context ^name clean-hallway) <context>)
  -(hallway ^restroom {> 0} ^section {> 0})
-->
  (remove <context>))

(p 401.wash-sheet
  ((context ^name laundry)
  ((dirty-sheet ^floor <floor> ^quantity {<ds> > 7}) <dirty-sheet>)
  ((washed-sheet ^floor <floor> ^quantity <ws>) <washed-sheet>)
-->
  (modify <dirty-sheet> ^quantity (compute <ds> - 8) )
  (modify <washed-sheet> ^quantity (compute <ws> + 8) ))

(p 401.wash-pillow-case
  ((context ^name laundry)
  ((dirty-pillow-case ^floor <floor>
    ^quantity {<dpc> > 7}) <dirty-pillow-case>)
  ((washed-pillow-case ^floor <floor>
    ^quantity <wpc>) <washed-pillow-case>)
-->
  (modify <dirty-pillow-case> ^quantity (compute <dpc> - 8) )
  (modify <washed-pillow-case> ^quantity (compute <wpc> + 8) ))

(p 402.dry-sheet
  ((context ^name laundry)
  ((washed-sheet ^floor <floor> ^quantity {<ws> > 7}) <washed-sheet>)
  ((clean-sheet ^floor <floor> ^quantity <cs>) <clean-sheet>)
-->
  (modify <washed-sheet> ^quantity (compute <ws> - 8) )
  (modify <clean-sheet> ^quantity (compute <cs> + 8) ))

(p 402.dry-pillow-case
  ((context ^name laundry)
```

```

((washed-pillow-case ^floor <floor>
  ^quantity (<wpc> > 7)) <washed-pillow-case>)
((clean-pillow-case ^floor <floor> ^quantity <cpc>) <clean-pillow-case>)
-->
(modify <washed-pillow-case> ^quantity (compute <wpc> - 8) )
(modify <clean-pillow-case> ^quantity (compute <cpc> + 8) )

(p 403.wash-towel
  (context ^name laundry)
  ((dirty-towel ^floor <floor> ^quantity (<dt> > 47)) <dirty-towel>)
  ((washed-towel ^floor <floor> ^quantity <wt> ) <washed-towel>)
-->
(modify <dirty-towel> ^quantity (compute <dt> - 48) )
(modify <washed-towel> ^quantity (compute <wt> + 48))

(p 404.dry-towel
  (context ^name laundry)
  ((washed-towel ^floor <floor> ^quantity (<wt> > 47)) <washed-towel>)
  ((clean-towel ^floor <floor> ^quantity <ct> ) <clean-towel>)
-->
(modify <washed-towel> ^quantity (compute <wt> - 48) )
(modify <clean-towel> ^quantity (compute <ct> + 48))

(p 437.done-laundry
  ((context ^name laundry) <context>)
-->
(remove <context>)

; there is a cross-product in this production it can be eliminated
; by using the ordering on the name and room-num fields.
(p 438.new-reservation
  (context ^name reservation)
  ((reservation ^name <n> ^arrival-date <ad>
    ^departure-date <dd>) <reservation>)
  ((room ^status vacant ^room-num <r>) <room>)
  -(room ^status vacant ^room-num > <r>)
-->
(modify <room> ^status reserved ^name <n>)
(remove <reservation>))

(p 439.excess-reservation
  (context ^name reservation)
  ((reservation) <reservation>)
  -(room ^status vacant)
-->
(remove <reservation>))

(p 446 done-reservation

```

```

((context ^name reservation) <context>)
-(reservation ^status new)
-->
(modify <context> ^name reservation-done))

(p 447.with-reservation
  (context ^name check-in)
  (date ^day <d>)
  ((customer ^name <n> ^status new
    ^arrival-date <d> ^departure-date <dd> ) <customer>)
  ((room ^room-num <rn> ^name <n> ^status reserved) <room>)
-->
(modify <customer> ^status staying ^room-num <rn>)
(modify <room> ^status occupied ^departure-date <dd>))

(p 448.without-reservation
  (context ^name check-in)
  (date ^day <d>)
  ((customer ^status new ^name <n>
    ^arrival-date <d> ^departure-date <dd> ) <customer>)
  -(room ^status reserved ^name <n>)
  ((room ^room-num <rn> ^status vacant) <room>)
  -(room ^room-num > <rn> ^status vacant)
-->
(modify <customer> ^status staying ^room-num <rn>)
(modify <room> ^status occupied ^departure-date <dd>))

(p 455.done-check-in
  ((context ^name check-in) <context>)
  -(customer ^status new)
-->
(remove <context>))

(p 488.set-menu-chowder-full
  (context ^name conference)
  (order-chowder ^conference <conf> ^quantity <q>)
  ((conference-chowder ^conference <conf> ^quantity 0) <conference>)
  ((soup ^name chowder ^quantity {<q1> >= <q>} ) <soup>)
-->
(modify <conference> ^quantity <q>)
(modify <soup> ^quantity (compute <q1> - <q>)))

(p 488.set-menu-chowder-partial
  (context ^name conference)
  (order-chowder ^conference <conf> ^quantity <q>)
  ((conference-chowder ^conference <conf> ^quantity 0) <conference>)
  ((soup ^name chowder ^quantity {<q1> < <q>} ) <soup>)
-->

```

```

(modify <conference> ^quantity <q1>)
(modify <soup> ^quantity 0))

(p 489.set-menu-chicken-full
  (context ^name conference)
  (order-chicken ^conference <conf> ^quantity <q>)
  {<conference-chicken ^conference <conf> ^quantity 0> <conference>}
  {<soup ^name chicken ^quantity {<q1> >= <q>}> <soup>})
-->
(modify <conference> ^quantity <q>)
(modify <soup> ^quantity (compute <q1> - <q>))

(p 489.set-menu-chicken-partial
  (context ^name conference)
  (order-chicken ^conference <conf> ^quantity <q>)
  {<conference-chicken ^conference <conf> ^quantity 0> <conference>}
  {<soup ^name chicken ^quantity {<q1> < <q>}> <soup>})
-->
(modify <conference> ^quantity <q1>)
(modify <soup> ^quantity 0)

(p 490.set-menu-hamburger-full
  (context ^name conference)
  (order-hamburger ^conference <conf> ^quantity <q>)
  {<conference-hamburger ^conference <conf> ^quantity 0> <conference>}
  {<dish ^name hamburger ^quantity {<q1> >= <q>}> <dish>})
-->
(modify <conference> ^quantity <q>)
(modify <dish> ^quantity (compute <q1> - <q>))

(p 490.set-menu-hamburger-partial
  (context ^name conference)
  (order-hamburger ^conference <conf> ^quantity <q>)
  {<conference-hamburger ^conference <conf> ^quantity 0> <conference>}
  {<dish ^name hamburger ^quantity {<q1> < <q>}> <dish>})
-->
(modify <conference> ^quantity <q1>)
(modify <dish> ^quantity 0)

(p 492.set-menu-steak-full
  (context ^name conference)
  (order-steak ^conference <conf> ^quantity <q>)
  {<conference-steak ^conference <conf> ^quantity 0> <conference>}
  {<dish ^name steak ^quantity {<q1> >= <q>}> <dish>})
-->
(modify <conference> ^quantity <q>)
(modify <dish> ^quantity (compute <q1> - <q>!)))

```

```

(p 492.set-menu-steak-partial
  (context ^name conference)
  (order-steak ^conference <conf> ^quantity <q>)
  {((conference-steak ^conference <conf> ^quantity 0) <conference>)
  {((dish ^name steak ^quantity (<ql> < <qp>)) <dish>)
-->
  (modify <conference> ^quantity <ql>)
  (modify <dish> ^quantity 0))

(p 498.set-menu-seafood-full
  (context ^name conference)
  (order-seafood ^conference <conf> ^quantity <q>)
  {((conference-seafood ^conference <conf> ^quantity 0) <conference>)
  {((dish ^name seafood ^quantity (<ql> >= <qp>)) <dish>)
-->
  (modify <conference> ^quantity <q>)
  (modify <dish> ^quantity (compute <ql> - <qp>)))

(p 498.set-menu-seafood-partial
  (context ^name conference)
  (order-seafood ^conference <conf> ^quantity <q>)
  {((conference-seafood ^conference <conf> ^quantity 0) <conference>)
  {((dish ^name seafood ^quantity (<ql> < <qp>)) <dish>)
-->
  (modify <conference> ^quantity <ql>)
  (modify <dish> ^quantity 0))

(p 493.set-menu-salad-full
  (context ^name conference)
  (order-salad ^conference <conf> ^quantity <q>)
  {((conference-salad ^conference <conf> ^quantity 0) <conference>)
  {((veg ^name salad ^quantity (<ql> >= <qp>)) <veg>)
-->
  (modify <conference> ^quantity <qp>)
  (modify <veg> ^quantity (compute <ql> - <qp>)))

(p 493.set-menu-salad-partial
  (context ^name conference)
  (order-salad ^conference <conf> ^quantity <q>)
  {((conference-salad ^conference <conf> ^quantity 0) <conference>)
  {((veg ^name salad ^quantity (<ql> < <qp>)) <veg>)
-->
  (modify <conference> ^quantity <ql>)
  (modify <veg> ^quantity 0))
(p 495.set-menu-fruit-full
  (context ^name conference)
  (order-fruit ^conference <conf> ^quantity <q>)
  {((conference-fruit ^conference <conf> ^quantity 0) <conference>)

```

```

((veg ^name fruit ^quantity {<q1> >= <q>}) <veg>)
-->
(modify <conference> ^quantity <q>
  (modify <veg> ^quantity (compute <q1> - <q>)))
(p 495.set-menu-fruit-partial
  (context ^name conference)
  (order-fruit ^conference <conf> ^quantity <q>)
  {<conference-fruit ^conference <conf> ^quantity 0>} <conference>
  {<veg ^name fruit ^quantity {<q1> < <q>}>} <veg>)
-->
(modify <conference> ^quantity <q1>)
(modify <veg> ^quantity 0)

(p 499.done-set-menu
  {<context ^name conference>} <context>)
-->
(remove <context>)
(make context ^name food-soup)
(make context ^name food-salad-fruit)
(make context ^name food-dish)
(make context ^name brew-coffee)
(make context ^name table))

(p 505.clean-table
  (context ^name table)
  {<table ^number <n> ^clean no>} <table>)
  {<table-cloth ^table-number <n>>} <cloth>)
  {<table-plate ^table-number <n>>} <plate>)
  {<table-knife ^table-number <n>>} <knife>)
  {<table-fork ^table-number <n>>} <fork>)
  {<table-napkin ^table-number <n>>} <napkin>)
-->
(modify <table> ^clean yes)
(modify <cloth> ^number 0)
(modify <plate> ^number 0)
(modify <knife> ^number 0)
(modify <fork> ^number 0)
(modify <napkin> ^number 0))

(p 500.set-chair
  (context ^name table)
  {<table ^number <table-id> ^clean yes>})
  {<table-chairs ^table-number <table-id> ^number 0>} <table-chairs>)
  {<chair ^number {<n> >= 4}>} <chair>)
-->
(modify <table-chairs> ^number 4)
(modify <chair> ^number (compute <n> - 4)))

```

```
(p 541.put-table-cloth
  (context ^name table)
  (table ^number <table-id> ^clean yes)
  ((table-cloth ^table-number <table-id> ^number 0) <table-cloth>)
  ((cloth ^number (<n> > 0)) <cloth>)
-->
  (modify <table-cloth> ^number 1)
  (modify <cloth> ^number (compute <n> - 1)))

(p 546.put-plate
  (context ^name table)
  (table ^number <table-id> ^clean yes)
  (table-cloth ^table-number <table-id> ^number 1)
  ((table-plate ^table-number <table-id> ^number 0) <table-plate>)
  ((plate ^number (<n> >= 4)) <plate>)
-->
  (modify <table-plate> ^number 4)
  (modify <plate> ^number (compute <n> - 4)))

(p 551.put-knife
  (context ^name table)
  (table ^number <table-id> ^clean yes)
  (table-cloth ^table-number <table-id> ^number 1)
  ((table-knife ^table-number <table-id> ^number 0) <table-knife>)
  ((knife ^number (<n> >= 4)) <knife>)
-->
  (modify <table-knife> ^number 4)
  (modify <knife> ^number (compute <n> - 4)))

(p 556.put-fork
  (context ^name table)
  (table ^number <table-id> ^clean yes)
  (table-cloth ^table-number <table-id> ^number 1)
  ((table-fork ^table-number <table-id> ^number 0) <table-fork>)
  ((fork ^number (<n> >= 4)) <fork>)
-->
  (modify <table-fork> ^number 4)
  (modify <fork> ^number (compute <n> - 4)))

(p 561.put-napkin
  (context ^name table)
  (table ^number <table-id> ^clean yes)
  (table-cloth ^table-number <table-id> ^number 1)
  ((table-napkin ^table-number <table-id> ^number 0) <table-napkin>)
  ((napkin ^number (<n> >= 4)) <napkin>)
-->
  (modify <table-napkin> ^number 4)
```

```
(modify <napkin> ^number (compute <n> - 4))

(p 641.done-table
  ((context ^name table) <context>)
-->
  (remove <context>))

(p 642.need-new-brew
  (context ^name brew-coffee)
  {((coffee-machine ^coffee old) <machine>)
-->
  (modify <machine> ^power off ^pot off))

(p 643.empty-beans
  (context ^name brew-coffee)
  {((coffee-machine ^power <> on ^beans (<> new <> empty)) <machine>)
-->
  (modify <machine> ^beans empty))

(p 644.add-new-beans
  (context ^name brew-coffee)
  {((coffee-machine ^power <> on ^beans empty) <machine>)
-->
  (modify <machine> ^beans new))

(p 645.add-pot
  (context ^name brew-coffee)
  {((coffee-machine ^power <> on
    ^pot <> on) <machine>)
-->
  (modify <machine> ^pot on))

(p 646.add-water
  (context ^name brew-coffee)
  {((coffee-machine ^power <> on
    ^water-level <> full) <machine>)
-->
  (modify <machine> ^water-level full) )

(p 647.brew-coffee
  (context ^name brew-coffee)
  {((coffee-machine ^power <> on ^beans new ^pot on
    ^water-level full) <machine>)
-->
  (modify <machine> ^power on ^coffee new))

(p 690.done-brewing-coffee
  ((context ^name brew-coffee) <brew>)
```

```
-->
  (remove <brew>))

(p 691 dish-hamburger
  (context ^name food-dish)
  { (dish-counter-hamburger ^conference <conf>
      ^quantity {<h> < 10}) <counter>
    { (conference-hamburger ^conference <conf>
      ^quantity {<h1> > 0}) <conference-dish> }

-->
  (modify <counter> ^quantity (compute <h> + 1))
  (modify <conference-dish> ^quantity (compute <h1> - 1)))

(p 692.d_sh-steak
  (context ^name food-dish)
  { (dish-counter-steak ^conference <conf>
      ^quantity {<s> < 10}) <counter>;
    { (conference-steak ^conference <conf>
      ^quantity {<s1> > 0}) <conference-dish> }

-->
  (modify <counter> ^quantity (compute <s> + 1))
  (modify <conference-dish> ^quantity (compute <s1> - 1)))

(p 693.dish-seafood
  (context ^name food-dish)
  { (dish-counter-seafood ^conference <conf>
      ^quantity {<s> < 10}) <counter>
    { (conference-seafood ^conference <conf>
      ^quantity {<s1> > 0}) <conference-dish> }

-->
  (modify <counter> ^quantity (compute <s> + 1))
  (modify <conference-dish> ^quantity (compute <s1> - 1)))

(p 703.done-dish
  { (context ^name food-dish) <context> }

-->
  (remove <context>))

(p 704.salad
  (context ^name food-salad-fruit)
  { (salad-bar ^conference <conf> ^quantity {<s> < 10}) <bar>
    { (conference-salad ^conference <conf>
      ^quantity {<s1> > 0}) <conference-veg> }

-->
  (modify <bar> ^quantity (compute <s> + 1))
  (modify <conference-veg> ^quantity (compute <s1> - 1)))

(p 705.fruit
```

```

(context ^name food-salad-fruit)
((fruit-bar ^conference <conf> ^quantity (<f> < 10)) <bar>
 { (conference-fruit ^conference <conf>
   ^quantity (<f1> > 0)) <conference-veg> }
-->
  (modify <bar> ^quantity (compute <f> + 1))
  (modify <conference-veg> ^quantity (compute <f1> - 1)))

(p 712.done-salad-fruit
  ((context ^name food-salad-fruit) <context>)
-->
  (remove <context>))

(p 713.soup-chowder
  (context ^name food-soup)
  (conference-chowder ^conference <conf> ^quantity > 0)
  { (soup-bar ^conference <conf> ^name chowder ^status empty) <soup> }
-->
  (modify <soup> ^status full))

(p 714.soup-chicken
  (context ^name food-soup)
  (conference-chicken ^conference <conf> ^quantity > 0)
  { (soup-bar ^conference <conf> ^name chicken ^status empty) <soup> }
-->
  (modify <soup> ^status full))

(p 721.done-soup
  ((context ^name food-soup) <context>)
-->
  (remove <context>))

(p 722.synchronization
  (context ^name done-maid-cleaning)
  /context ^name reservation-done)
-->
  (make context ^name check-in))

```

E.8 Parallel version of hotel

```

; Program to model the operation of a hotel.
;
; Original version by: Steve Kuo, University of Southern California
;
; Parallelized by: Anurag Acharya, acha2cs.cmu.edu

```

```
-----  
; The original version had a fixed number of floors and had applied  
; copy and constraint very extensively. So much so that C compilers  
; refused to compile files that large. There are several other  
; optimizations including splitting room-linen and room-furniture  
; tuples so that the affect set size goes down.  
;  
-----  
; Tuple declarations  
;  
(literalize customer name status reservation arrival-date  
    departure-date num-person room-num cash)  
(literalize room room-num status phase departure-date room-type  
    floor name)  
(literalize hallway floor restroom section)  
(literalize rate room-type num-person rate)  
(literalize reservation status name num-person  
    arrival-date departure-date)  
(literalize room-towel room-num number)  
(literalize room-sheet room-num number)  
(literalize room-pillow-case room-num number)  
(literalize room-trash-can room-num status)  
(literalize room-blanket room-num status)  
(literalize room-bedsheet room-num status)  
(literalize room-bathroom room-num status)  
(literalize room-dresser room-num status)  
(literalize room-table room-num status)  
(literalize room-chairs room-num status)  
(literalize room-vacuum room-num status)  
(literalize dirty-sheet floor quantity)  
(literalize washed-sheet floor quantity)  
(literalize clean-sheet floor quantity)  
(literalize dirty-towel floor quantity)  
(literalize washed-towel floor quantity)  
(literalize clean-towel floor quantity)  
(literalize dirty-pillow-case floor quantity)  
(literalize washed-pillow-case floor quantity)  
(literalize clean-pillow-case floor quantity)  
(literalize soup name quantity)  
(literalize dish name quantity)  
(literalize veg name quantity)  
(literalize order-chowder conference quantity)  
(literalize order-chicken conference quantity)  
(literalize order-hamburger conference quantity)  
(literalize order-steak conference quantity)  
(literalize order-seafood conference quantity)  
(literalize order-salad conference quantity)
```

```
(literalize order-fruit conference quantity)
(literalize conference-chowder conference quantity)
(literalize conference-chicken conference quantity)
(literalize conference-hamburger conference quantity)
(literalize conference-steak conference quantity)
(literalize conference-seafood conference quantity)
(literalize conference-salad conference quantity)
(literalize conference-fruit conference quantity)
(literalize table conference number clean)
(literalize table-cloth table-number number)
(literalize table-plate table-number number)
(literalize table-knife table-number number)
(literalize table-fork table-number number)
(literalize table-napkin table-number number)
(literalize table-chairs table-number number)
(literalize chair number)
(literalize cloth number)
(literalize plate number)
(literalize fork number)
(literalize knife number)
(literalize napkin number)
(literalize dish-counter-hamburger conference number quantity)
(literalize dish-counter-steak conference number quantity)
(literalize dish-counter-seafood conference number quantity)
(literalize salad-bar conference number quantity)
(literalize fruit-bar conference number quantity)
(literalize soup-bar conference name status)
(literalize coffee-machine conference number power coffee
    beans pot water-level)
(literalize context name)
(literalize date day)

(p 0.initialize
  (start)
-->
  (make context ^name clean-hallway)
  (make context ^name conference)
  (make context ^name check-out)
  (make context ^name reservation)
  (make context ^name check-in))

(p 73.done-check-out
  { (context ^name check-out) <context> }
-->
  (modify <context> ^name maid-laundry))

(p 146.done-maid-laundry
  { (context ^name maid-laundry) <context> }
```

```
-(room ^phase maid-laundry)
-->
(modify <context> ^name maid-cleaning))

(p 381.done-maid-cleaning
 {((context ^name maid-cleaning) <context>)
 -(room ^phase maid-cleaning)
-->
 (remove <context>)
 (make context ^name laundry))

(p 437.done-laundry
 {((context ^name laundry) <context>,
 -(dirty-pillow-case ^quantity > 7)
 -(dirty-sheet ^quantity 7)
 -(dirty-towel ^quantity > 47)
-->
 (remove <context>))

(p 400.done-clean-hallway
 {((context ^name clean-hallway) <context>)
 -(hallway ^restroom (> 0) ^section (> 0))
-->
 (remove <context>))

(p 446.done-reservation
 {((context ^name reservation) <context>)
 -(reservation ^status new)
-->
 (remove <context>))

(p 455.done-check-in
 {((context ^name check-in) <context>)
 -(customer ^status new)
-->
 (remove <context>))

(p done-menu
 {((context ^name conference) <context>)
 {((done-task ^name chowder) <menu1>)
 {((done-task ^name chicken) <menu2>)
 {((done-task ^name hamburger) <menu3>)
 {((done-task ^name seafood) <menu4>)
 {((done-task ^name salad) <menu5>)
 {((done-task ^name fruit) <menu6>)
-->
 (remove <context> <menu1> <menu2> <menu3> <menu4> <menu5>
 <menu6>)
```

```

(make context ^name clean-table)
(make context ^name brew-coffee)
(make context ^name food-dish)
(make context ^name food-salad-fruit)
(make context ^name food-soup)

(p done-clean-table
  ((context ^name clean-table) <context>)
  -->
  (modify <context> ^name set-table))

(p 641.done-table
  ((context ^name set-table) <context>)
  ((done-task ^name table-cloth) <task1>)
  ((done-task ^name plate) <task2>)
  ((done-task ^name knife) <task3>)
  ((done-task ^name fork) <task4>)
  ((done-task ^name napkin) <task5>)
  -->
  (remove <context> <task1> <task2> <task3> <task4> <task5>))

(p 703.done-dish
  ((context ^name food-dish) <context>)
  ((done-task ^name hamburger) <task1>)
  ((done-task ^name steak) <task2>)
  ((done-task ^name seafood) <task3>)
  -->
  (remove <context> <task1> <task2> <task3>))

(p done-food-salad-fruit
  ((context ^name food-salad-fruit) <context>)
  ((done-task ^name salad-bar) <task1>)
  ((done-task ^name fruit-bar) <task2>)
  -->
  (remove <context> <task1> <task2>))

(p 721.done-soup
  ((context ^name food-soup) <context>)
  ((done-task ^name chowder) <task1>)
  ((done-task ^name chicken) <task2>)
  -->
  (remove <context> <task1> <task2>))

(pset leaving-customer
(parp 1.check-out

```

```

(context ^name check-out)
(date ^day <d>)
{ (customer ^status staying ^arrival-date <ad> ^departure-date <dc>
           ^num-person <np> ^room-num <r> ^cash <c>) <customer> }
{ (room ^room-num <r> ^room-type <rt> ^status occupied) <room> }
{ (rate ^room-type <rt> ^num-person <np> ^rate <rate>) }
{ (room-towel ^room-num <r>) <towel> }
{ (room-sheet ^room-num <r>) <sheet> }
{ (room-pillow-case ^room-num <r>) <pillow-case> }
{ (room-trash-can ^room-num <r>) <trash-can> }
{ (room-blanket ^room-num <r>) <blanket> }
{ (room-bedspread ^room-num <r>) <bed-spread> }
{ (room-bathroom ^room-num <r>) <bathroom> }
{ (room-dresser ^room-num <r>) <dresser> }
{ (room-table ^room-num <r>) <table> }
{ (room-chairs ^room-num <r>) <chairs> }
{ (room-vacuum ^room-num <r>) <vacuum> }

-->
(modify <customer> ^status check-out
          ^cash (compute <c> - (<rate> * (<d> - <ad>))))
(modify <room> ^status change ^phase maid-laundry)
(modify <towel> ^number 8)
(modify <sheet> ^number 4)
(modify <pillow-case> ^number 4)
(modify <trash-can> ^status not-empty)
(modify <blanket> ^status not-made)
(modify <bed-spread> ^status not-made)
(modify <bathroom> ^status dirty)
(modify <dresser> ^status dirty)
(modify <table> ^status dirty)
(modify <chairs> ^status dirty)
(modify <vacuum> ^status dirty))

{ pset staying-customer
(parp 19.staying-over
(context ^name check-out)
(date ^day <d>)
{ (customer ^status staying ^room-num <r>
           ^departure-date (<dd> > <dc>) <customer> )
{ (room ^room-num <r> ^status occupied) <room> }
{ (room-towel ^room-num <r>) <towel> }
{ (room-sheet ^room-num <r>) <sheet> }
{ (room-pillow-case ^room-num <r>) <pillow-case> }
{ (room-trash-can ^room-num <r>) <trash-can> }
{ (room-blanket ^room-num <r>) <blanket> }
{ (room-bedspread ^room-num <r>) <bed-spread> }
{ (room-bathroom ^room-num <r>) <bathroom> }
{ (room-dresser ^room-num <r>) <dresser> }

```

```

((room-table ^room-num <r>) <table>)
((room-chairs ^room-num <r>) <chairs>)
((room-vacuum ^room-num <r>) <vacuum>)
-->
(modify <room> ^status make-up ^phase maid-cleaning)
(modify <towel> ^number 0)
(modify <sheet> ^number 0)
(modify <pillow-case> ^number 0)
(modify <trash-can> ^status not-empty)
(modify <blanket> ^status not-made)
(modify <bed-spread> ^status not-made)
(modify <bathroom> ^status dirty)
(modify <dresser> ^status dirty)
(modify <table> ^status dirty)
(modify <chairs> ^status dirty)
(modify <vacuum> ^status dirty)))

(pset strip-towels
(p 74.strip-towel
  (context ^name maid-laundry)
  (room ^room-num <r> ^phase maid-laundry ^floor <floor>)
  ((room-towel ^room-num <r> ^number (<t> > 0)) <room-linen>)
  ((dirty-towel ^floor <floor> ^quantity <q>) <towel>)
-->
  (modify <room-linen> ^number 0)
  (modify <towel> ^quantity (compute <q> + <t>)))

(pset strip-sheet
(p 75.strip-sheet
  (context ^name maid-laundry)
  (room ^room-num <r> ^phase maid-laundry ^floor <floor>)
  ((room-sheet ^room-num <r> ^number (<s> > 0)) <room-linen>)
  ((dirty-sheet ^floor <floor> ^quantity <q>) <sheet>)
-->
  (modify <room-linen> ^number 0)
  (modify <sheet> ^quantity (compute <q> + <s>)))

(pset strip-pillow-case
(p 76.strip-pillow-case
  (context ^name maid-laundry)
  (room ^room-num <r> ^phase maid-laundry ^floor <floor>)
  ((room-pillow-case ^room-num <r> ^number (<p> > 0)) <room-linen>)
  ((dirty-pillow-case ^floor <floor> ^quantity <q>) <case>)
-->
  (modify <room-linen> ^number 0)
  (modify <case> ^quantity (compute <q> + <p>)))

(pset finish-laundry-room

```

```
(parp 77.finish-laundry-room
  (context ^name maid-laundry)
  { (room ^room-num <r> ^phase maid-laundry) <room> }
  { (room-towel ^room-num <r> ^number 0)
  { (room-sheet ^room-num <r> ^number 0)
  { (room-pillow-case ^room-num <r> ^number 0)
  -->
  (modify <room> ^phase maid-cleaning)))

(pset towel-needed
(parp 147.towel-needed
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  { (room-towel ^room-num <r> ^number < 8) <room-linen> }
  -->
  (modify <room-linen> ^number 8)))

(pset make-sheet
(parp 148.make-sheet
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  { (room-sheet ^room-num <r> ^number < 4) <room-linen> }
  -->
  (modify <room-linen> ^number 4)))

(pset make-pillow
(parp 149.make-pillow
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  { (room-pillow-case ^room-num <r> ^number < 4) <room-linen> }
  -->
  (modify <room-linen> ^number 4)))

(pset clean-trash-can
(parp 150.clean-trash-can
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  { (room-trash-can ^room-num <r> ^status not-empty) <room-linen> }
  -->
  (modify <room-linen> ^status empty)))

(pset make-blanket
(parp 151.make-blanket
  (context ^name maid-cleaning)
  (room ^room-num <r> ^phase maid-cleaning)
  { (room-blanket ^room-num <r> ^status not-made) <room-linen> }
  -->
  (modify <room-linen> ^status made)))
```

```
(pset make-bed-spread
  (parp 152.make-bed-spread
    (context ^name maid-cleaning)
    (room ^room-num <r> ^phase maid-cleaning)
    ((room-bedspread ^room-num <r> ^status not-made) <room-linen>)
-->
  (modify <room-linen> ^status made)))

(pset clean-bathroom
  (parp 153.clean-bathroom
    (context ^name maid-cleaning)
    (room ^room-num <r> ^phase maid-cleaning)
    ((room-bathroom ^room-num <r> ^status dirty) <room-furniture>)
-->
  (modify <room-furniture> ^status clean)))

(pset clean-dresser
  (parp 154.clean-dresser
    (context ^name maid-cleaning)
    (room ^room-num <r> ^phase maid-cleaning)
    ((room-dresser ^room-num <r> ^status dirty) <room-furniture>)
-->
  (modify <room-furniture> ^status clean)))

(pset clean-table-room
  (parp 155.clean-table
    (context ^name maid-cleaning)
    (room ^room-num <r> ^phase maid-cleaning)
    ((room-table ^room-num <r> ^status dirty) <room-furniture>)
-->
  (modify <room-furniture> ^status clean)))

(pset clean-chairs
  (parp 156.clean-chairs
    (context ^name maid-cleaning)
    (room ^room-num <r> ^phase maid-cleaning)
    ((room-chairs ^room-num <r> ^status dirty) <room-furniture>)
-->
  (modify <room-furniture> ^status clean)))

(pset vacuum
  (parp 157.vacuum
    (context ^name maid-cleaning)
    (room ^room-num <r> ^phase maid-cleaning)
    ((room-vacuum ^room-num <r> ^status dirty) <room-furniture>)
-->
  (modify <room-furniture> ^status done)))
```

```

(pset finish-room-changing
(parp 158.finish-room-changing
  (context ^name maid-cleaning)
  {((room ^room-num <r> ^status change ^phase maid-cleaning) <room>)
  (room-towel ^room-num <r> ^number 8)
  (room-sheet ^room-num <r> ^number 4)
  (room-pillow-case ^room-num <r> ^number 4)
  (room-trash-can ^room-num <r> ^status empty)
  (room-blanket ^room-num <r> ^status made)
  (room-bedspread ^room-num <r> ^status made)
  (room-bathroom ^room-num <r> ^status clean)
  (room-dresser ^room-num <r> ^status clean)
  (room-table ^room-num <r> ^status clean)
  (room-chairs ^room-num <r> ^status clean)
  (room-vacuum ^room-num <r> ^status done)
-->
  (modify <room> ^status vacant ^phase clean)
  })

(pset finish-room-makeup
(parp 159.finish-room-make-up
  (context ^name maid-cleaning)
  {((room ^room-num <r> ^status make-up ^phase maid-cleaning) <room>)
  (room-towel ^room-num <r> ^number 8)
  (room-sheet ^room-num <r> ^number 4)
  (room-pillow-case ^room-num <r> ^number 4)
  (room-trash-can ^room-num <r> ^status empty)
  (room-blanket ^room-num <r> ^status made)
  (room-bedspread ^room-num <r> ^status made)
  (room-bathroom ^room-num <r> ^status clean)
  (room-dresser ^room-num <r> ^status clean)
  (room-table ^room-num <r> ^status clean)
  (room-chairs ^room-num <r> ^status clean)
  (room-vacuum ^room-num <r> ^status done)
-->
  (modify <room> ^status occupied ^phase clean))}

(pset launder-sheet
(p 401.launder-sheet
  (context ^name laundry)
  {((dirty-sheet ^floor <floor> ^quantity (<ds> > 7)) <dirty-sheet>)
  {((clean-sheet ^floor <floor> ^quantity <cs>) <clean-sheet>)
-->
  (modify <dirty-sheet> ^quantity (compute <ds> - 8) )
  (modify <clean-sheet> ^quantity (compute <cs> + 8) ))}

(pset launder-pillow-case

```

```

(p 401.launder-pillow-case
  (context ^name laundry)
  {((dirty-pillow-case ^floor <floor>
    ^quantity (<dpc> > 7)) <dirty-pillow-case>)
   {((clean-pillow-case ^floor <floor>
    ^quantity <cpc>) <clean-pillow-case>)
  -->
    (modify <dirty-pillow-case> ^quantity (compute <dpc> - 8) )
    (modify <clean-pillow-case> ^quantity (compute <cpc> + 8) ))}

(pset launder-towels
(p 405.launder-towel
  (context ^name laundry)
  {((dirty-towel ^floor <floor> ^quantity (<dt> > 47)) <dirty-towel>)
   {((clean-towel ^floor <floor> ^quantity <ct>) <clean-towel>)
  -->
    (modify <dirty-towel> ^quantity (compute <dt> - 48) )
    (modify <clean-towel> ^quantity (compute <ct> + 48)))}

(pset clean-restroom
(p 382.clean-restroom
  (context ^name clean-hallway)
  {((hallway ^restroom (<n> > 0)) <hallway>)
  -->
    (modify <hallway> ^restroom (compute <n> - 1)))}

(pset reservation
(p 438.new-reservation
  (context ^name reservation)
  {((reservation ^name <n> ^arrival-date <ad>
    ^departure-date <dd>) <reservation>)
   {((room ^status vacant ^room-num <r>) <room>)
    -(room ^status vacant ^room-num > <r>)
  -->
    (modify <room> ^status reserved ^name <n>)
    (remove <reservation>))

(p 448.without-reservation
  (context ^name check-in)
  (date ^day <d>)
  {((customer ^status new ^name <n> ^reservation no
    ^arrival-date <d> ^departure-date <dd>) <customer>)
   {((room ^room-num <rn> ^status vacant) <room>)
    -(room ^room-num > <rn> ^status vacant)
  -->
    (modify <customer> ^status staying ^room-num <rn>)
    (modify <room> ^status occupied ^departure-date <dd>))}
```

```

(pset excess-reservation
  (parp 439.excess-reservation
    (context ^name done-maid-cleaning)
    ((reservation ^name <n>) <reservation>)
    -(room ^status vacant)
-->
    (remove <reservation>))

(pset with-reservation
  (parp 447.with-reservation
    (context ^name check-in)
    (date ^day <d>)
    ((customer ^name <n> ^status new ^reservation yes
      ^arrival-date <d> ^departure-date <dd> ) <customer>)
    ((room ^room-num <rn> ^name <n> ^status reserved) <room>)
-->
    (modify <customer> ^status staying ^room-num <rn>)
    (modify <room> ^status occupied ^departure-date <dd>))

(pset chowder-menu
  (p 488.set-menu-chowder-full
    (context ^name conference)
    (order-chowder ^conference <conf> ^quantity <q>)
    ((conference-chowder ^conference <conf> ^quantity 0) <conference>)
    ((soup ^name chowder ^quantity {<q1> >= <q>} ) <soup>)
-->
    (modify <conference> ^quantity <q>)
    (modify <soup> ^quantity (compute <q1> - <q>)))

  (p 488.set-menu-chowder-partial
    (context ^name conference)
    (order-chowder ^conference <conf> ^quantity <q>)
    ((conference-chowder ^conference <conf> ^quantity 0) <conference>)
    ((soup ^name chowder ^quantity {<q1> < <q>} ) <soup>)
-->
    (modify <conference> ^quantity <q1>)
    (modify <soup> ^quantity 0))

  (p done-chowder-menu
    (context ^name conference)
-->
    (make done-task ^name chowder)))

(pset chicken-menu
  (p 489.set-menu-chicken-full
    (context ^name conference)
    (order-chicken ^conference <conf> ^quantity <q>)
    ((conference-chicken ^conference <conf> ^quantity 0) <conference>))

```

```

((soup ^name chicken ^quantity {<ql> >= <q>}) <soup>)
-->
(modify <conference> ^quantity <q>
(modify <soup> ^quantity (compute <ql> - <q>)))

(p 489.set-menu-chicken-partial
  (context ^name conference)
  (order-chicken ^conference <conf> ^quantity <q>
  ((conference-chicken ^conference <conf> ^quantity 0) <conference>)
  ((soup ^name chicken ^quantity {<ql> < <q>}) <soup>)
-->
(modify <conference> ^quantity <ql>)
(modify <soup> ^quantity 0))

(p done-chicken-menu
  (context ^name conference)
  -->
  (make done-task ^name chicken)))

(pset hamburger-menu
(p 490.set-menu-hamburger-full
  (context ^name conference)
  (order-hamburger ^conference <conf> ^quantity <q>
  ((conference-hamburger ^conference <conf> ^quantity 0) <conference>)
  ((dish ^name hamburger ^quantity {<ql> >= <q>}) <dish>)
-->
(modify <conference> ^quantity <q>)
(modify <dish> ^quantity (compute <ql> - <q>)))

(p 490.set-menu-hamburger-partial
  (context ^name conference)
  (order-hamburger ^conference <conf> ^quantity <q>
  ((conference-hamburger ^conference <conf> ^quantity 0) <conference>)
  ((dish ^name hamburger ^quantity {<ql> < <q>}) <dish>)
-->
(modify <conference> ^quantity <ql>)
(modify <dish> ^quantity 0))

(p done-hamburger-menu
  (context ^name conference)
  -->
  (make done-task ^name hamburger)))

(pset steak-menu
(p 492.set-menu-steak-full
  (context ^name conference)
  (order-steak ^conference <conf> ^quantity <q>
  ((conference-steak ^conference <conf> ^quantity 0) <conference>)

```

```

((dish ^name steak ^quantity (<q1> >= <q>)) <dish>)
-->
(modify <conference> ^quantity <q>)
(modify <dish> ^quantity (compute <q1> - <q>))

(p 492.set-menu-steak-partial
  (context ^name conference)
  (order-steak ^conference <conf> ^quantity <q>)
  ((conference-steak ^conference <conf> ^quantity 0) <conference>)
  ((dish ^name steak ^quantity (<q1> < <q>)) <dish>)
-->
(modify <conference> ^quantity <q1>)
(modify <dish> ^quantity 0))
(p done-steak-menu
  (context ^name conference)
  -->
  (make done-task ^name steak))

(pset seafood-menu
(p 498.set-menu-seafood-full
  (context ^name conference)
  (order-seafood ^conference <conf> ^quantity <q>)
  ((conference-seafood ^conference <conf> ^quantity 0) <conference>)
  ((dish ^name seafood ^quantity (<q1> >= <q>)) <dish>)
-->
(modify <conference> ^quantity <q>)
(modify <dish> ^quantity (compute <q1> - <q>)))

(p 498.set-menu-seafood-partial
  (context ^name conference)
  (order-seafood ^conference <conf> ^quantity <q>)
  ((conference-seafood ^conference <conf> ^quantity 0) <conference>)
  ((dish ^name seafood ^quantity (<q1> < <q>)) <dish>)
-->
(modify <conference> ^quantity <q1>)
(modify <dish> ^quantity 0))

(p done-seafood-menu
  (context ^name conference)
  -->
  (make done-task ^name seafood))

(pset salad-menu
(p 493.set-menu-salad-full
  (context ^name conference)
  (order-salad ^conference <conf> ^quantity <q>)
  ((conference-salad ^conference <conf> ^quantity 0) <conference>)
  ((veg ^name salad ^quantity (<q1> >= <q>)) <veg>)

```

```

-->
(modify <conference> ^quantity <q>)
(modify <veg> ^quantity (compute <q1> - <q>))

(p 493.set-menu-salad-partial
  (context ^name conference)
  (order-salad ^conference <conf> ^quantity <q>)
  ((conference-salad ^conference <conf> ^quantity 0) <conference>)
  ((veg ^name salad ^quantity <q1> <q>) <veg>)

-->
(modify <conference> ^quantity <q1>)
(modify <veg> ^quantity 0))
(p done-salad-menu
  (context ^name conference)
  -->
  (make done-task ^name salad)))

(pset fruit-menu
(p 495.set-menu-fruit-full
  (context ^name conference)
  (order-fruit ^conference <conf> ^quantity <q>)
  ((conference-fruit ^conference <conf> ^quantity 0) <conference>)
  ((veg ^name fruit ^quantity (<q1> >= <q>)) <veg>)

-->
(modify <conference> ^quantity <q>)
(modify <veg> ^quantity (compute <q1> - <q>))
(p 495.set-menu-fruit-partial
  (context ^name conference)
  (order-fruit ^conference <conf> ^quantity <q>)
  ((conference-fruit ^conference <conf> ^quantity 0) <conference>)
  ((veg ^name fruit ^quantity (<q1> < q)) <veg>)

-->
(modify <conference> ^quantity <q1>)
(modify <veg> ^quantity 0))
(p done-fruit-menu
  (context ^name conference)
  -->
  (make done-task ^name fruit)))

(pset clean-table-conf
(parp 505.clean-table
  (context ^name clean-table)
  ((table ^number <n> ^clean no) <table>)
  ((table-cloth ^table-number <n>) <cloth>)
  ((table-plate ^table-number <n>) <plate>)
  ((table-knife ^table-number <n>) <knife>)
  ((table-fork ^table-number <n>) <fork>)
  ((table-napkin ^table-number <n>) <napkin>)

```

```

-->
  (modify <table> ^clean yes)
  (modify <cloth> ^number 0)
  (modify <plate> ^number 0)
  (modify <knife> ^number 0)
  (modify <fork> ^number 0)
  (modify <napkin> ^number 0))

(pset set-chair
(p 500.set-chair
  (context ^name set-table)
  (table ^number <table-id> ^clean yes)
  {<table-chairs ^table-number <table-id> ^number 0; <table-chairs>}
  {<chair ^number (<n> >= 4); <chair>}
-->
  (modify <table-chairs> ^number 4)
  (modify <chair> ^number (compute <n> - 4)))

(p done-chairs
  (context ^name set-table)
  -->
  (make done-task ^name chairs))

(pset put-table-cloth
(p 541.put-table-cloth
  (context ^name set-table)
  (table ^number <table-id> ^clean yes)
  {<table-cloth ^table-number <table-id> ^number 0; <table-cloth>}
  {<cloth ^number (<n> > 0); <cloth>}
-->
  (modify <table-cloth> ^number 1)
  (modify <cloth> ^number (compute <n> - 1)))
(p done-table-cloth
  (context ^name set-table)
  -->
  (make done-task ^name table-cloth))

(pset put-plate
(p 546.put-plate
  (context ^name set-table)
  (table ^number <table-id> ^clean yes)
  {<table-cloth ^table-number <table-id> ^number 1; <table-plate>}
  {<table-plate ^table-number <table-id> ^number 0; <table-plate>}
  {<plate ^number (<n> >= 4); <plate>}
-->
  (modify <table-plate> ^number 4)
  (modify <plate> ^number (compute <n> - 4)))

```

```

(p done-plate
  (context ^name set-table)
  -->
  (make done-task ^name plate))

(pset put-knife
(p 551.put-knife
  (context ^name set-table)
  (table ^number <table-id> ^clean yes)
  (table-cloth ^table-number <table-id> ^number 1)
  ((table-knife ^table-number <table-id> ^number 0) <table-knife>)
  ((knife ^number (<n> >= 4)) <knife>)
  -->
  (modify <table-knife> ^number 4)
  (modify <knife> ^number (compute <n> - 4)))
(p done-knife
  (context ^name set-table)
  -->
  (make done-task ^name knife))

(pset put-fork
(p 556.put-fork
  (context ^name set-table)
  (table ^number <table-id> ^clean yes)
  (table-cloth ^table-number <table-id> ^number 1)
  ((table-fork ^table-number <table-id> ^number 0) <table-fork>)
  ((fork ^number (<n> >= 4)) <fork>)
  -->
  (modify <table-fork> ^number 4)
  (modify <fork> ^number (compute <n> - 4)))
(p done-fork
  (context ^name set-table)
  -->
  (make done-task ^name fork))

(pset put-napkin
(p 561.put-napkin
  (context ^name set-table)
  (table ^number <table-id> ^clean yes)
  (table-cloth ^table-number <table-id> ^number 1)
  ((table-napkin ^table-number <table-id> ^number 0) <table-napkin>)
  ((napkin ^number (<n> >= 4)) <napkin>)
  -->
  (modify <table-napkin> ^number 4)
  (modify <napkin> ^number (compute <n> - 4)))
(p done-napkin
  (context ^name set-table)
  -->

```

```

        (make done-task ^name napkin)))

(pset dish-hamburger
(p 691.dish-hamburger
  (context ^name food-dish)
  ((dish-counter-hamburger ^conference <conf>
    ^quantity :<h> < 10) <counter>)
  ((conference-hamburger ^conference <conf>
    ^quantity (<h1> > 0)) <conference-dish>)
-->
  (modify <counter> ^quantity (compute <h> + 1))
  (modify <conference-dish> ^quantity (compute <h1> - 1)))

(p done-dish-hamburger
  (context ^name food-dish)
  -->
  (make done-task dish-hamburger))

(pset dish-steak
(p 692.dish-steak
  (context ^name food-dish)
  ((dish-counter-steak ^conference <conf>
    ^quantity (<s> < 10) <counter>)
  ((conference-steak ^conference <conf>
    ^quantity (<s1> > 0)) <conference-dish>)
-->
  (modify <counter> ^quantity (compute <s> + 1))
  (modify <conference-dish> ^quantity (compute <s1> - 1)))
(p done-dish-steak
  (context ^name food-dish)
  -->
  (make done-task dish-steak))

(pset dish-seafood
(p 693.dish-seafood
  (context ^name food-dish)
  ((dish-counter-seafood ^conference <conf>
    ^quantity (<s> < 10) <counter>)
  ((conference-seafood ^conference <conf>
    ^quantity (<s1> > 0)) <conference-dish>)
-->
  (modify <counter> ^quantity (compute <s> + 1))
  (modify <conference-dish> ^quantity (compute <s1> - 1)))
(p done-dish-seafood
  (context ^name food-dish)
  -->
  (make done-task dish-seafood)))

```

```

(pset salad
(p 704.salad
  (context ^name food-salad-fruit)
  { (salad-bar ^conference <conf> ^quantity {<s> < 10}) <bar> }
  { (conference-salad ^conference <conf>
    ^quantity {<s1> > 0}) <conference-veg> )
  -->
    (modify <bar> ^quantity (compute <s> + 1))
    (modify <conference-veg> ^quantity (compute <s1> - 1)))
(p done-salad-dish
  (context ^name food-salad-fruit)
  -->
  (make done-task ^name salad-bar))

(pset fruit
(p 705.fruit
  (context ^name food-salad-fruit)
  { (fruit-bar ^conference <conf> ^quantity {<f> < 10}) <bar> }
  { (conference-fruit ^conference <conf>
    ^quantity {<f1> > 0}) <conference-veg> )
  -->
    (modify <bar> ^quantity (compute <f> + 1))
    (modify <conference-veg> ^quantity (compute <f1> - 1)))
(p done-fruit-dish
  (context ^name food-salad-fruit)
  -->
  (make done-task ^name fruit-bar))

(pset chowder-dish
(p 713.soup-chowder
  (context ^name food-soup)
  (conference-chowder ^conference <conf> ^quantity > 0)
  { (soup-bar ^conference <conf> ^name chowder ^status empty) <soup> }
  -->
  (modify <soup> ^status full))

(p done-chowder
  (context ^name food-soup)
  -->
  (make done-task ^name chowder))

(pset chicken-dish
(p 714.soup-chicken
  (context ^name food-soup)
  (conference-chicken ^conference <conf> ^quantity > 0)
  { (soup-bar ^conference <conf> ^name chicken ^status empty) <soup> }
  -->

```

```
(modify <soup> ^status full))
(p done-chicken
  (context ^name food-soup)
  -->
  (make done-task ^name chicken))

(pset brew-coffee
(parp 642.need-new-brew
  (context ^name brew-coffee)
  { (coffee-machine ^coffee old) <machine>}
  -->
  (modify <machine> ^power off ^pot off))

(parp 643.empty-beans
  (context ^name brew-coffee)
  { (coffee-machine ^power <> on ^beans {<> new <> empty}) <machine>}
  -->
  (modify <machine> ^beans empty))

(parp 644.add-new-beans
  (context ^name brew-coffee)
  { (coffee-machine ^power <> on ^beans empty) <machine>}
  -->
  (modify <machine> ^beans new))

(parp 645.add-pot
  (context ^name brew-coffee)
  { (coffee-machine ^power <> on
    ^pot <> on) <machine>}
  -->
  (modify <machine> ^pot on))

(parp 646.add-water
  (context ^name brew-coffee)
  { (coffee-machine ^power <> on
    ^water-level <> full) <machine>}
  -->
  (modify <machine> ^water-level full) )

(parp 647.brew-coffee
  (context ^name brew-coffee)
  { (coffee-machine ^power <> on ^beans new ^pot on
    ^water-level full ^conference <conf>
    ^number <number>) <machine>}
  -->
  (modify <machine> ^power on ^coffee new))

(p 690.done-brewing-coffee
```

```
((context ^name brew-coffee) <brew>)
-->
(remove <brew>)))
```

Appendix F

Code for benchmarks used in COM experiments

This appendix contains the source code for the three benchmarks used in the collection-oriented match experiments. Two versions are presented for every program – a tuple-oriented version written in PPL and a collection-oriented version written in COPL

F.1 Tuple-oriented version of make-teams

```
-----  
; This program counts the total number of "high-quality"  
; teams that can be assembled from a pool of individuals  
; different specializations. The quality of the team  
; is measured as a simple function of the merit points of  
; each individual  
; written by: Milind Tambe, Carnegie Mellon University  
; converted to PPL: Anurag Acharya, Carnegie Mellon University  
-----  
  
(literalize person id area previous-project merit-points)  
(literalize goal type)  
(literalize count value)  
(literalize team hardware operating-systems networks compilers  
    merit-value select-status)  
  
(p start  
    (start)  
-->  
    (make goal ^type create-team))
```

```

(p change-goal-type-1
  ((goal ^type create-team) <cond1>
-->
  (modify <cond1> ^type select-team))

(p change-goal-type-2
  ((goal ^type select-team) <cond1>
-->
  (modify <cond1> ^type count-teams))

(p count-teams-start
  (goal ^type count-teams)
  (team ^select-status selected)
  -(count)
-->
  (make count ^value 0))

(p count-teams-2
  (goal ^type count-teams)
  ((team ^select-status selected) <cond2>
  ( (count ^value <value>) <cond>
-->
  (modify <cond> ^value (compute (<value> + 1)))
  (modify <cond2> ^select-status counted))

(p change-goal-type-3
  ((goal ^type count-teams) <cond1>
-->
  (modify <cond1> ^type print-value))

(p print-value
  (goal ^type print-value)
  (count ^value <value>)
-->
  (write (crlf) value is <value> (crlf)))

(pset create-teams
(parp make-team
  (goal ^type create-team)
  (person ^id <id1> ^area hardware ^previous-project <p1>
  ^merit-points <v1>)
  (person ^id <id2> ^area operating-systems
  ^previous-project <p2> ^merit-points <v2>)
  (person ^id <id3> ^area networks ^previous-project <p3>
  ^merit-points <v3>)
  (person ^id <id4> ^area compilers ^previous-project <p1>
  ^merit-points <v4>)

```

```
-->
  (make team ^hardware <id1> ^operating-systems <id2>
          ^networks <id3> ^compilers <id4>
          ^select-status unknown
          ^merit-value (compute (<v1>+<v2>+<v3>+<v4>))))}

(pset filter-teams
(parp select-teams-1
  (goal ^type select-team)
  { <cond2> (team ^merit-value > 8 ^select-status unknown)})
-->
  (modify <cond2> ^select-status selected))}
```

F.2 Collection-oriented version of make-teams

```
-----
; This program counts the total number of "high-quality"
; teams that can be assembled from a pool of individuals
; different specializations. The quality of the team
; is measured as a simple function of the merit points of
; each individual
; Converted to COPL: Anurag Acharya, Carnegie Mellon University
;-----
(external compute4)
(external cardinality)
(literalize person id area merit-points previous-project)
(literalize goal type)
(literalize count value)
(literalize team hardware operating-systems compilers
            networks merit-value select-status)

(p start
  (start)
-->
  (insert (make goal ^type create-team)))

(p make-team
  (goal ^type create-team)
  (person ^id <id1> ^area hardware
         ^previous-project <p1> ^merit-points <v1>)
  (person ^id <id2> ^area operating-systems
         ^previous-project <p2> ^merit-points <v2>)
  (person ^id <id3> ^area networks
         ^previous-project <p3> ^merit-points <v3>)
  (person ^id <id4> ^area compilers
         ^previous-project <p4> ^merit-points <v4>)
```

```

-->
  (insert (update (make team ^select-status unknown ^hardware <id1>
                           ^operating-systems <id2> ^networks <id3>
                           ^compilers <id4>)
                  ^merit-value (compute4 <v1> <v2> <v3> <v4>)))))

(p change-goal-type-1
  ((goal ^type create-team) <cond1>)
-->
  (modify <cond1> ^type select-team))

(p select-teams
  (goal ^type select-team)
  { (team ^merit-value > 8 ^select-status unknown) <cond2>)
-->
  (modify <cond2> ^select-status selected))

(p change-goal-type-2
  ((goal ^type select-team) <cond1>)
-->
  (modify <cond1> ^type count-teams))

(p count-teams-start
  (goal ^type count-teams)
  (team ^select-status selected ^hardware <id>)
-->
  (insert (make count ^value (cardinality <id>)))))

(p change-goal-type-3
  ((goal ^type count-teams) <cond1>)
-->
  (modify <cond1> ^type print-value))

(p print-value
  (goal ^type print-value)
  (count ^value <value>)
-->
  (write (crlf) value is <value> (crlf)))

```

F.3 Tuple-oriented version of clusters

; this program was motivated from a high level vision expert system.
; It calculates the distance between different objects in a scene,
; given the co-ordinates of the objects. It then takes one object

```
; as the center point and groups together objects that are close
; to it (ie within a certain range of distance) The SPAM expert
; system uses a similar computation in its LCC and
; functional-area phase (although in a much more complex
; fashion)
; written by: Milind Tambe, Carnegie Mellon University
; Converted to PPL: Anurag Acharya, Carnegie Mellon University
;-----
(literalize object number x y focus type)
(literalize distance object1 object2 value)
(literalize group center member counted)
(literalize group-count center size counted)
(literalize average-size sum total average)
(literalize goal name)

(p start
  (start)
-->
  (make goal ^name calculate-distance))

(p s2
  ((goal ^name calculate-distance) <c1>)
-->
  (modify <c1> ^name create-foci))

(p s4
  ((goal ^name create-foci) <c1>)
-->
  (modify <c1> ^name create-groups))

(p s6
  ((goal ^name create-groups) <c1>)
-->
  (modify <c1> ^name get-group-sizes))

(p s7
  (goal ^name get-group-sizes)
  (object ^number <n1> ^focus yes)
-->
  (make group-count ^center <r1> ^size 0 ^counted no))

(p s8
  (goal ^name get-group-sizes)
  {(group-count ^center <n1> ^size <num>) <c1>}
  {(group ^center <n1> ^member <n2> ^counted no) <c2>}
-->
  (modify <c1> ^size (compute (<num> + 1)))
  (modify <c2> ^counted yes))
```

```

• (p s9
  ((goal ^name get-group-sizes) <c3>)
-->
  (modify <c3> ^name average-group-sizes)
  (make average-size ^sum 0 ^total 0))

• (p s10
  (goal ^name average-group-sizes)
  { (average-size ^sum <s> ^total <t>) <c2> }
  { (group-count ^center <n1> ^size <sz> ^counted no) <c1> }
-->
  (modify <c2> ^sum (compute (<s> + <sz>))
           ^total (compute (<t> + 1)))
  (modify <c1> ^counted yes))

• (p s11
  (goal ^name average-group-sizes)
  { (average-size ^sum <sum> ^total (<> 0 <total>) ^average nil) <c2> }
-->
  (modify <c2> ^average (compute (<sum> / <total>)))

• (p s12
  (goal ^name average-group-sizes)
  (average-size ^sum <sum> ^total <total> ^average <ave>)
-->
  (write (crlf) average is <ave> (crlf)))

(pset find-distance
• (parp s1
  (goal ^name calculate-distance)
  (object ^number <n1> ^x <x> ^y <y>
          ^type <<tarmac parking-apron hangar-building>>)
  (object ^number <n2> ^x <x1> ^y <y1>)
-->
  (make distance ^object1 <n1> ^object2 <n2>
           ^value (compute ((<x1>-<x>)*(<x1>-<x>)+(<y1>-<y>)*(<y1>-<y>)))))

• (pset create-seeds
• (parp s3
  (goal ^name create-foci)
  { (object ^number <n2> ^focus no
          ^type <<tarmac parking-apron hangar-building>>) <c1> }
-->
  (modify <c1> ^focus yes)))

```

```

(pset make-areas
(parp s5
  (goal ^name create-groups)
  (object ^number <n1> ^focus yes)
  (distance ^object1 <n1> ^object2 <n2> ^value { > 0 < 800 })
-->
  (make group ^center <n1> ^member <n2> ^counted no)))

```

F.4 Collection-oriented version of clusters

```

;-----;
; this program was motivated from a high level vision expert system.
; It calculates the distance between different objects in a scene,
; given the co-ordinates of the objects. It then takes one object
; as the center point and groups together objects that are close
; to it (ie within a certain range of distance) The SPAM expert
; system uses a similar computation in its LCC and
; functional-area phase (although in a much more complex
; fashion)
; Converted to COPL: Anurag Acharya, Carnegie Mellon University
;-----;
(external distancecompute)
(external bigsum)
(external divcompute)
(external cardinality)
(literalize object number x y focus type)
(literalize distance object1 object2 value)
(literalize group center member counted)
(literalize group-count center size counted)
(literalize average-size sum total average)
(literalize goal name)

(p start
  (start)
-->
  (insert (make goal ^name calculate-distance));

(p s1
  (goal ^name calculate-distance)
  (object ^number <n1> ^x <x> ^y <y>
    ^type <<armac parking-apron hangar-building>>)
  (object ^number <n2> ^x <x1> ^y <y1>)
-->
  (insert (update (make distance ^object1 <n1> ^object2 <n2>)
    ^value (distancecompute <x1> <x> <y1> <y>))))
```

```

(p s2
  ((goal ^name calculate-distance) <c1>)
-->
  (modify <c1> ^name create-foci))

(p s3
  ((goal ^name create-foci)
   ((object ^number <n2> ^focus no
            ^type <<armac parking-apron hangar-building>>) <c1>)
-->
  (modify <c1> ^focus yes))

(p s4
  ((goal ^name create-foci) <c1>)
-->
  (modify <c1> ^name create-groups))

(p s5
  ((goal ^name create-groups)
   (object ^number <n1> ^focus yes)
   (distance ^object1 <n1> ^object2 <n2> ^value ( > 0 < 800 )))
-->
  (insert (make group ^center <n1> ^member <n2> ^counted no)))

(p s6
  ((goal ^name create-groups) <c1>)
-->
  (modify <c1> ^name get-group-sizes))

(p s7-modified
  ((goal ^name get-group-sizes)
   (object ^number <n1> ^focus yes)
   (group ^center <n1> ^member <n2> ^counted no)
-->
  (insert (make group-count ^center <n1>
                ^size (cardinality <n2>) ^counted no)))

(p s9
  ((goal ^name get-group-sizes) <c3>)
-->
  (modify <c3> ^name average-group-sizes))

(p s10
  ((goal ^name average-group-sizes)
   (group-count ^center <n1> ^size <sz> ^counted no) <c1>)
-->
  (insert (make average-size ^sum (bigsum <sz>
                                         ^total (cardinality <n1>) ^average 0 ))))

```

```

(p s11
  (goal ^name average-group-sizes)
  ((average-size ^sum <sum> ^total {> 0 <t>} ^average 0) <c2>)
-->
  (modify <c2> ^average (divcompute <sum> <t> )))

(p s12
  (goal ^name average-group-sizes)
  (average-size ^sum <sum> ^total <total> ^average <ave>)
-->
  (write (crlf) average is <ave> (crlf)))

```

F.5 Tuple-oriented version of airline-route

```

;-----;
; This program operates on a flight database and finds the
; minimum cost route for a desired source and destination
; , written by: Milind Tambe, Carnegie Mellon University
; , Converted to PPL: Anurag Acharya, Carnegie Mellon University
;-----;
(literalize goal name)
(literalize route id cost flight1 flight2 flight3 flight4 flight5
  length for)
(literalize mincost for length recommended cost)
(literalize flight source destination airline time id cost)
(literalize traveller name source destination)
(literalize travel-constraint name hop-constraint time-constraint)

(p initialize
  (start)
-->
  (make goal ^name get-traveller-information))

(p get-traveller-information
  (goal ^name get-traveller-information)
  (goal ^name get-traveller-information)
-->
  (make traveller ^name Anurag ^source Pittsburgh
    ^destination Jhumritaliyya)
  (make travel-constraint ^name Anurag ^hop-constraint 3))

(p change-goal-type
  (goal ^name get-traveller-information) <c1>)
-->
  (modify <c1> ^name compute-routes))

```

```

, currently we will only deal with hop constraint
(p print-routes
  { <c1> (goal ^name compute-routes)}
-->
  (modify <c1> ^name print-routes))

, the following production uses specificity to ensure
; firing it. it is a hack
(p print-lowest-route-cost
  (goal ^name print-routes)
  (goal ^name print-routes)
  (goal ^name print-routes)
  (travel-constraint ^name <x> ^hop-constraint <length> )
  (route ^id <route> ^length <length> ^for <x> ^cost <c>)
  -(route ^id <route2> ^length <length> ^for <x> ^cost < <c>)
-->
  (make mincost ^cost <c> ^for <x> ^length <length> ^recommended yes))

(p delete-if-hop-constraint-satisfied
  (goal ^name print-routes)
  (goal ^name print-routes)
  (goal ^name print-routes)
  { (travel-constraint ^name <x> ^hop-constraint <length> <d>)
    (route ^id <route> ^length <length> ^for <x>)
    (mincost ^cost <c> ^for <x> ^length <length> ^recommended yes)
-->
  (remove <d>)}

(p no-constrained-routes-min-cost
  (goal ^name print-routes)
  (goal ^name print-routes)
  (travel-constraint ^name <x> ^hop-constraint <length>)
  (route ^id <route> ^length <length1> ^for <x> ^cost <c>)
  -(route ^id <route2> ^length <length2> ^for <x> ^cost { < <c> })
-->
  (make mincost ^cost <c> ^for <x> ^length <length> ^recommended no))

(p delete-if-hop-constraint-satisfied-2
  (goal ^name print-routes)
  { (travel-constraint ^name <x> ^hop-constraint <length> ) <d>)
    (route ^id <route> ^length <length2> ^for <x>)
    (mincost ^cost <c> ^for <x> ^length <length> ^recommended no)
-->
  (remove <d>)}

```

```

(p change-goal-value-to-print-cost
  ((goal ^name print-routes) <c> )
-->
  (modify <c> ^name print-cost))

(p print-cost-1
  (goal ^name print-cost)
  (mincost ^cost <c> ^length <length> ^for <x> ^recommended no)
-->
  (write mincost undesired route for <x> has cost <c> (crlf)))

(p print-cost-2
  (goal ^name print-cost)
  (mincost ^cost <c> ^length <length> ^for <x> ^recommended yes)
-->
  (write (crlf) mincost desired route for <x> has cost <c>))

(pset direct-routes
(parp hop1
  (goal ^name compute-routes)
  (traveller ^name <x> ^source <source> ^destination <dest>)
  (flight ^source <source> ^destination <dest> ^cost <cost> ^id <id>)
-->
  (make route ^length 1 ^id (genatom) ^for <x> ^flight1 <id> ^cost <cost>))
)

(pset one-stops
(parp hop2
  (goal ^name compute-routes)
  (traveller ^name <x> ^source <source> ^destination <dest>)
  (flight ^source <source> ^destination <intmd> ^cost <cost1> ^id <id1>)
  (flight ^source <intmd> ^destination <dest> ^cost <cost2> ^id <id2>)
-->
  (make route ^length 2 ^id (genatom) ^for <x> ^flight1 <id1>
        ^flight2 <id2> ^cost (compute <cost1> + <cost2>)))
)

(pset two-stops
(parp hop3
  (goal ^name compute-routes)
  (traveller ^name <x> ^source <source> ^destination <dest>)
  (flight ^source <source> ^destination <intmd> ^cost <cost1> ^id <id1>)
  (flight ^source <intmd> ^destination <intmd2> ^cost <cost2> ^id <id2>)
  (flight ^source <intmd2> ^destination <dest> ^cost <cost3> ^id <id3>)
-->
  (make route ^length 3 ^id (genatom) ^for <x> ^flight1 <id1>
        ^flight2 <id2> ^flight3 <id3>

```

```

        ^cost (compute <cost1> + <cost2> + <cost3>)))
    }

(pset three-stops
(parp hop4
    (goal  ^name compute-routes)
    (traveller ^name <x> ^source <source> ^destination <dest>
    (flight ^source <source> ^destination <intmd> ^cost <cost1> ^id <id1>
    (flight ^source <intmd> ^destination <intmd2> ^cost <cost2> ^id <id2>
    (flight ^source <intmd2> ^destination <intmd3> ^cost <cost3> ^id <id3>
    (flight ^source <intmd3> ^destination <dest> ^cost <cost4> ^id <id4>
-->
    (make route ^length 4 ^id (genatom) ^for <x> ^flight1 <id1>
        ^flight2 <id2> ^flight3 <id3> ^flight4 <id4>
    ^cost (compute <cost1> + <cost2> + <cost3> + <cost4>)))
}

(pset four-stops
(parp hop5
    (goal  ^name compute-routes)
    (traveller ^name <x> ^source <source> ^destination <dest>
    (flight ^source <source> ^destination <intmd> ^cost <cost1> ^id <id1>
    (flight ^source <intmd> ^destination <intmd2> ^cost <cost2> ^id <id2>
    (flight ^source <intmd2> ^destination <intmd3> ^cost <cost3> ^id <id3>
    (flight ^source <intmd3> ^destination <intmd4> ^cost <cost4> ^id <id4>
    (flight ^source <intmd4> ^destination <dest> ^cost <cost5> ^id <id5>
-->
    (make route ^length 5 ^id (genatom) ^for <x> ^flight1 <id1>
        ^flight2 <id2> ^flight3 <id3> ^flight4 <id4>
        ^flight5 <id5>
    ^cost (compute <cost1>+<cost2>+<cost3>+<cost4>+<cost5>)))
}

```

F.6 Collection-oriented version of airline-route

```

-----
; This program operates on a flight database and finds the
; minimum cost route for a desired source and destination
; Converted to COPL: Anurag Acharya, Carnegie Mellon University
-----
(external compute2)
(external compute3)
(external compute4)
(external compute5)
(external min)
(external cardinality)

```

```

(literalize goal name)
(literalize route id cost flight1 flight2 flight3 flight4
  flight5 length for)
(literalize mincost for length recommended cost)
(literalize flight id source destination time airline cost)
(literalize traveller name source destination)
(literalize travel-constraint name hop-constraint time-constraint)

(p initialize
  (start)
-->
  (insert (make goal ^name get-traveller-information)))

(p get-traveller-information
  (goal ^name get-traveller-information)
  (goal ^name get-traveller-information)
-->
  (insert (make traveller ^name Anurag ^source Pittsburgh
    ^destination Jhumrataliyya))
  (insert (make travel-constraint ^name Anurag ^hop-constraint 3)))

(p change-goal-type
  ((goal ^name get-traveller-information) <c1>)
-->
  (modify <c1> ^name compute-routes))

(p hop1
  (goal ^name compute-routes)
  (goal ^name compute-routes)
  (traveller ^name <x> ^source <source> ^destination <dest>)
  (flight ^source <source> ^destination <dest> ^cost <cost> ^id <id>)
-->
  (insert (update (make route ^length 1 ^for <x> ^flight1 <id>)
    ^cost <cost>
    ^id (vector-genatom (cardinality <id>)))))

(p hop2
  (goal ^name compute-routes)
  (traveller ^name <x> ^source <source> ^destination <dest>)
  (flight ^source <source> ^destination <intmd> ^cost <cost1> ^id <id1>)
  (flight ^source <intmd> ^destination <dest> ^id <id2> ^cost <cost2>)
-->
  (insert (update (make route ^length 2 ^for <x> ^flight1 <id1>
    ^flight2 <id2>)
    ^cost (compute2 <cost1> <cost2>)
    ^id (vector-genatom (cardinality <id1> *
      (cardinality <id2>)))))

(p hop3
  (goal ^name compute-routes)

```

```

(traveller ^name <x> ^source <source> ^destination <dest>
(flight ^source <source> ^destination <intmd> ^cost <cost1> ^id <id1>
(flight ^source <intmd> ^destination <intmd2> ^cost <cost2> ^id <id2>
(flight ^source <intmd2> ^destination <dest> ^cost <cost3> ^id <id3>
-->
(insert (update (make route ^length 3 ^for <x> ^flight1 <id1>
^flight2 <id2> ^flight3 <id3>
^cost (compute3 <cost1> <cost2> <cost3>)
^id (vector-genatom (cardinality <id1>) *
(cardinality <id2>) *
(cardinality <id3>)))))

(p hop4
(goal ^name compute-routes)
(traveller ^name <x> ^source <source> ^destination <dest>
(flight ^source <source> ^destination <intmd> ^cost <cost1> ^id <id1>
(flight ^source <intmd> ^destination <intmd2> ^cost <cost2> ^id <id2>
(flight ^source <intmd2> ^destination <intmd3> ^cost <cost3> ^id <id3>
(flight ^source <intmd3> ^destination <dest> ^id <id4> ^cost <cost4>
-->
(insert (update (make route ^length 4 ^for <x> ^flight1 <id1>
^flight2 <id2> ^flight3 <id3>
^flight4 <id4>
^cost (compute3 <cost1> <cost2> <cost3> <cost4>)
^id (vector-genatom (cardinality <id1>) *
(cardinality <id2>) *
(cardinality <id3>)
(cardinality <id4>)))))

(p hop5
(goal ^name compute-routes)
(traveller ^name <x> ^source <source> ^destination <dest>
(flight ^source <source> ^destination <intmd> ^cost <cost1> ^id <id1>
(flight ^source <intmd> ^destination <intmd2> ^cost <cost2> ^id <id2>
(flight ^source <intmd2> ^destination <intmd3> ^cost <cost3> ^id <id3>
(flight ^source <intmd3> ^destination <intmd4> ^cost <cost4> ^id <id4>
(flight ^source <intmd4> ^destination <dest> ^cost <cost5> ^id <id5>
-->
(insert (update (make route ^length 5 ^for <x> ^flight1 <id1>
^flight2 <id2> ^flight3 <id3>
^flight4 <id4> ^flight5 <id5>
^cost (compute3 <cost1> <cost2> <cost3> <cost4> <cost5>)
^id (vector-genatom (cardinality <id1>) *
(cardinality <id2>) *
(cardinality <id3>)
(cardinality <id4>)
(cardinality <id5>)))))

(p print-routes

```

```

  ((goal ^name compute-routes) <c1>
-->
  (modify <c1> ^name print-routes))

(p print-lowest-route-cost
  (goal ^name print-routes)
  (goal ^name print-routes)
  (goal ^name print-routes)
  (travel-constraint ^name <x> ^hop-constraint <length> )
  (route ^id <route> ^length <length> ^for <x> ^cost <c>)
-->
  (insert (make mincost ^cost (min <c>) ^for <x> ^length <length>
    ^recommended yes)))

(p delete-if hop-constraint-satisfied
  (goal ^name print-routes)
  (goal ^name print-routes)
  (goal ^name print-routes)
  ((travel-constraint ^name <x> ^hop-constraint <length> ) <d>)
  (route ^id <route> ^length <length> ^for <x>)
  (mincost ^cost <c> ^for <x> ^length <length> ^recommended yes)
-->
  (remove <d>))

(p no-constrained-routes-min-cost
  (goal ^name print-routes)
  (goal ^name print-routes)
  (travel-constraint ^name <x> ^hop-constraint <length> )
  (route ^id <route> ^length <length1> ^for <x> ^cost <c>)
-->
  (insert (make mincost ^cost (min <c>) ^for <x> ^length <length>
    ^recommended no)))

(p delete-if-hop-constraint-satisfied-2
  (goal ^name print-routes)
  ((travel-constraint ^name <x> ^hop-constraint <length> ) <d>)
  (route ^id <route> ^length <length2> ^for <x>)
  (mincost ^cost <c> ^for <x> ^length <length> ^recommended no)
-->
  (remove <d>))

(p change-goal-value-to-print-cost
  ((goal ^name print-routes) <c>)
-->
  (modify <c> ^name print-cost))

```

```
(p print-cost-1
  (goal ^name print-cost)
  (mincost ^cost <c> ^length <length> ^for <x> ^recommended no)
-->
  (write (crlf) mincost undesired route for <x> has cost <c> (crlf)))

• (p print-cost-2
  (goal ^name print-cost)
  (mincost `cost <c> ^length <length> ^for <x> ^recommended yes)
-->
  (write (crlf) mincost desired route for <x> has cost <c> (crlf)))
```

Bibliography

- [1] A. Acharya and D. Kalp. Release notes for CParaOPSS 5.3 and ParaOPSS 4.4. This is distributed with the CParaOPSS release on dravido.soar.cs.cmu.edu in /usr/nemo/cparaops5, 1990.
- [2] A. Acharya, M. Tambe, and A. Gupta. Implementations of production systems on message passing computers. *IEEE Transactions on Parallel and Distributed Computing*, 3(4):477-87, July 1992.
- [3] J. R. Anderson. *The Architecture of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1983.
- [4] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. In *Proceedings of PARLE'87, volume 2*, pages 1-29, June 1987.
- [5] J. Bein, R. King, and N. Kamel. MOBY: An architecture for distributed expert database systems In *Proceedings of the thirteenth International Conference on Very Large Databases*, pages 13-20, Sept 1987.
- [6] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. Technical Report CMU-CS-93-119, School of Computer Science, Carnegie Mellon University, 1993.
- [7] G. Blelloch. CIS: A massively parallel rule-based system. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 735-41, August 1986.
- [8] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.
- [9] A. Bonner and M. Kifer. Transactional logic programming. Technical Report CSRI-270, Computer Systems Research Institute, University of Toronto, December 1993.
- [10] E. Brewer, A. Colbrook, C. Dellarocas, and W. Weihl. PROTEUS: a high-performance parallel-architecture simulator. In *Proceedings of the ACM SIGMETRICS and Performance'92 Conference*, pages 247-8, June 1992.

- [11] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming* Addison-Wesley, Reading, Massachusetts, 1985.
- [12] B. Buchanan and E. Feigenbaum. DENDRAL and Meta-DENDRAL: Their application dimensions. *Artificial Intelligence*, 11(1):5-24, 1978.
- [13] P. Buneman and E. Clemons. Efficiently monitoring database systems. *ACM Transactions on Database Systems*, Sept 1979.
- [14] P. L. Butler, J. D. Allen, and D. W. Bouldin. Parallel architecture for OPS5. In *Proceedings of the Fifteenth International Symposium on Computer Architecture*, pages 452-7, 1988.
- [15] J.-P. Cheiney, G. Kiernan, and C. de Maindreville. A database rule language compiler supporting parallelism. Technical Report 1397, Institut National de la Recherche en Informatique et Automatique, February 1991.
- [16] F.-C. Cheng and M.-Y. Wu. DFLOPS: A data flow machine for production systems. Technical Report CUCS-025-93, Department of Computer Science, Columbia University, November 1993.
- [17] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, Heidelberg, 1981.
- [18] R. Davis. Reasoning about control. *Artificial Intelligence*, 15:179-222, 1980.
- [19] R. Davis, B. Buchanan, and E. Shortliffe. Production rules as a representation for knowledge-based consultation program. *Artificial Intelligence*, 8:15-45, 1977.
- [20] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1-38, 1988.
- [21] L. M. Delcambre, J. Waramahaputi, and J. N. Etheredge. Pattern match reduction for the relational production language in the USL MMDBS. *SIGMOD Record*, 18(3):59-67, September 1989.
- [22] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453-7, August 1978.
- [23] R. Doorenbos. Matching 100,000 rules. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 290-6, 1993.

- [24] K. Easwaran. Specification, implementation and interactions of a rule subsystem in an integrated database system. Technical report, IBM Research Report RJ1820, August 1976.
- [25] C. Forgy. Note on production systems and ILLIAC-IV. Technical Report CMU-CS-80-130, Department of Computer Science, Carnegie Mellon University, 1980.
- [26] C. Forgy and D. Phillips. *RAL Language Guide (version 1)*. Production System Technologies Inc., 1991.
- [27] C. L. Forgy. *On the Efficient Implementation of Production Systems*. PhD thesis, Computer Science Department, Carnegie Mellon University, February 1979.
- [28] C. L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Computer Science Department, Carnegie Mellon University, July 1981.
- [29] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17-37, 1982.
- [30] C. L. Forgy. The OPS83 report. Technical Report CMU-CS-84-133, Computer Science Department, Carnegie Mellon University, May 1984.
- [31] A. Forin, J. Barrera, and R. Sanzi. The shared memory server. In *Proceedings of USENIX - Winter 89*, pages 229-43, 1989.
- [32] E. Freudenthal and A. Gottlieb. Process coordination with fetch-and-increment. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 260-8, April 1991.
- [33] D. Gadbois and D. Miranker. Discovering procedural executions of rule-based programs. In *Proceedings of AAAI-94*, pages 459-64, 1994.
- [34] J.-L. Gaudiot, S. Lee, and A. Sohn. Data-driven multiprocessor implementation of the rete match algorithm. In *Proceedings of the International Conference on Parallel Processing*, pages 256-9, 1988.
- [35] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM3 Users's Guide and Reference Manual*. Oak Ridge National Laboratory, May 1993.
- [36] D. Gordin and A. Pasik. Set-oriented constructs: From Rete rule bases to database systems. In *International Conference on Management of Data (SIGMOD) 1991*, pages 60-7, May 1991.
- [37] A. Gupta. Implementing OPS5 production systems on DADO. Technical Report CMU-CS-84-115, Computer Science Department, Carnegie Mellon University, March 1984.

- [38] A. Gupta. *Parallelism in Production Systems*. PhD thesis, Computer Science Department, Carnegie Mellon University, March 1986.
- [39] A. Gupta and C. Forgy. Measurements on production systems. Technical Report CMU-CS-83-167, Computer Science Department, Carnegie Mellon University, December 1983.
- [40] A. Gupta, C. Forgy, A. Newell, and R. Wedig. Parallel algorithms and architectures for production systems. In *Proceedings of the Thirteenth International Symposium on Computer Architecture*, pages 28-35, June 1986.
- [41] A. Gupta, C. L. Forgy, D. Kalp, A. Newell, and M. Tambe. Parallel OPS5 on the Encore Multimax. In *Proceedings of the International Conference on Parallel Processing*, pages 271-80, August 1988.
- [42] A. Gupta, M. Tambe, , D. Kalp, C. Forgy, and A. Newell. Parallel implementation of OPS5 on the Encore multiprocessor. Results and analysis. *International Journal of Parallel Programming*, 17(2):95-124, April 1988.
- [43] A. Gupta and M. Tambe. Suitability of message passing computers for implementing production systems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 687-92, August 1988.
- [44] E. N. Hanson. The design and implementation of the Ariel active database rule system. Technical Report UF-CIS-018-92, Department of Computer and Information Sciences, University of Florida, September 1991.
- [45] E. N. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 49-58, June 1992.
- [46] E. N. Hanson and J. Widom. An overview of production rules in database systems. *Knowledge Engineering Review*, 8(2):121-43, June 1993.
- [47] W. Harvey. Personal communication, 1993.
- [48] W. Harvey, D. Kalp, M. Tambe, , D. McKeown, and A. Newell. Measuring the effectiveness of task-level parallelism for high-level vision. *Journal of Parallel and Distributed Computing*, 13(4):395-411, 1991.
- [49] F. Hayes-Roth and D. Mostow. An automatically compilable recognition network for structured patterns. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 246-51, 1975.

- [50] M. Hernandez and S. Stolfo. Parallel programming of rule-based systems in PARULEL. In *Proceedings of the IJCAI-93 Workshop on Production Systems and their Innovative Applications*, August 1993.
- [51] B. K. Hillyer and D. E. Shaw. Execution of OPS5 production systems on a massively parallel machine. *Journal of Parallel and Distributed Processing*, 3:236-268, 1986.
- [52] P. Hudak et al. Report on the programming language Haskell: A non-strict purely functional language (version 1.2). *SIGPLAN Notices*, May 1992.
- [53] *IBM Systems Application Architecture, Common Programming Interface: Database Reference*, October 1988. IBM Form Number SC26-4348-1.
- [54] INGRES Products Division, Alameda CA. *Ingres V6.3 Reference Manual*, 1990.
- [55] T. Ishida. Parallel rule firing in production systems. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):11-7, March 1991.
- [56] T. Ishida and S. Stolfo. Towards the parallel execution of rules in production system programs. In *Proceedings of the International Conference on Parallel Programming*, pages 568-74, August 1985.
- [57] J.E.Laird. *Universal Subgoaling*. PhD thesis, Computer Science Department, Carnegie Mellon University, June 1983. (Also available in *Universal Subgoaling and Chunking: The automatic generation of goal hierarchies*, Kluwer Academic Publishers, 1986.).
- [58] R. H. Jr. An assessment of Multilisp: lessons from experience. *International Journal of Parallel Programming*, 15(6):459-501, December 1986.
- [59] G. Kahn and J. McDermott. The MUD system. In *Proceedings of the First Conference on Artificial Intelligence Applications*, December 1985.
- [60] D. Kalp. Personal communication, 1991.
- [61] M. A. Kelly and R. A. Seviora. Performance of OPS5 matching on CUPID. *Microprocessing and Microprogramming*, 27:397-404, August 1989.
- [62] T. Kowalski and D. Thomas. The VLSI design automation assistant: Prototype system. In *Proceedings of the 20th Design Automation Conference*, June 1983.
- [63] D. Kranz, J. R.H. Halstead, and E. Mohr. Mul-T: a high-performance parallel lisp. In *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 81-90, June 1989.

- [64] C.-M. Kuo, D. Miranker, and J. Browne. On the performance of the CREL system. *Journal of Parallel and Distributed Computing*, 13(4):424-41, December 1991.
- [65] S. Kuo, D. Moldovan, and S. Cha. MCMR: a multiple rule firing production system model. In *Proceedings of the Fifth International Parallelism Processing Symposium*, pages 256-9, 1991.
- [66] J. Laird, A. Newell, , and P. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1-64, 1987.
- [67] R. Landau. Diagnosing faults at the Australia Telescope. In *Proceedings of the IJCAI-93 Workshop on Production Systems and their Innovative Applications*, August 1993.
- [68] J. Larus. Abstract execution: A technique for efficiently tracing programs. *Software - Practice and Experience*, 20(12):1241-58, Dec 1990.
- [69] S. Manchanda. *A Dynamic Logic Programming Language for relational updates*. PhD thesis, Department of Computer Science, State University of New York at Stony Brook, December 1987. (also available as technical report TR-88-2 from the Department of Computer Science, University of Arizona, Tucson).
- [70] R. McBride and K. Lynn. Anatomy of rule-based simulation. In *Proceedings of the SCS Multiconference 1990*, pages 24-9, January 1990.
- [71] D. McCracken. *A Production System Version of the HEARSAY-II Speech Understanding System*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1978.
- [72] J. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(2):39-88, 1982.
- [73] J. McDermott and C. Forgy. Production system conflict resolution strategies. *Pattern-directed Inference Systems*, pages 177-99, 1978.
- [74] J. McDermott, A. Newell, and J. Moore. The efficiency of certain production system implementations. *Pattern-directed Inference Systems*, pages 155-76, 1978.
- [75] D. McKeown, W. Harvey, and J. McDermott. Rule based interpretation of aerial imagery. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(5):570-85, 1985.
- [76] S. Minton. *Learning Effective Search Control Knowledge: An explanation-based approach*. PhD thesis, Computer Science Department, Carnegie Mellon University, March 1988.

- [77] D. P. Miranker. Performance estimates for the DADO machine: A comparison of Rete and Treat. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984.
- [78] D. P. Miranker. *TREAT. A New and Efficient Match Algorithm for AI Production Systems*. PhD thesis, Computer Science Department, Columbia University, 1987.
- [79] D. P. Miranker and D. A. Brant. An algorithmic basis for integrating production systems and large databases. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 353–60, February 1990.
- [80] D. P. Miranker, D. A. Brant, B. Lofaso, and D. Gadbois. On the performance of lazy matching in production systems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 685–92, August 1990.
- [81] D. P. Miranker, C.-M. Kuo, and J. C. Browne. Parallelizing transforms for a concurrent rule execution language. Technical Report TR-89-30, Department of Computer Science, University of Texas at Austin, October 1989.
- [82] D. P. Miranker and B. Lofaso. The organization and performance of a TREAT-based production system compiler. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):3–10, March 1991.
- [83] J. Miyazaki, K. Takeda, H. Amano, and H. Aiso. A new version of a parallel production system machine, MANJI - II. In *Proceedings of the Sixth International Workshop on Database Machines*, pages 317–30, June 1989.
- [84] D. I. Moldovan. RUBIC: A multiprocessor for rule-based systems. *IEEE Transactions on Systems, Man and Cybernetics*, 19(4):699–706, July/August 1989.
- [85] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 251–62, March 1988.
- [86] D. E. Neiman. Control issues in parallel rule-firing production systems. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 310–6, July 1991.
- [87] A. Newell. Production systems: Models of control structures. In W. G. Chase, editor, *Visual Information Processing*, pages 463–526. Academic Press, New York, New York, 1973.
- [88] A. Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.

- [89] A. Newell and H. Simon. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
- [90] K. Oflazer. *Partitioning in Parallel Processing of Production Systems*. PhD thesis, Computer Science Department, Carnegie Mellon University, March 1987.
- [91] A. Osiyanwo and P. Dasiewicz. A parallel model and architecture for production s. In *Proceedings of the International Conference on Parallel Processing*, pages 141-153, August 1987.
- [92] A. Pasik. A source 'o-source transformation for increasing rule-based system parallelism. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):336-43, August 1992.
- [93] A. J. Pasik. *A Methodology for Programming Production Systems and its Implications on Parallelism*. PhD thesis, Department of Computer Science, Columbia University, 1989.
- [94] M. Perlin and J.-M. Debaud. Match Box: Fine-grained parallelism at the match level. In *Proceedings of the IEEE International Workshop on Tools for Artificial Intelligence*, pages 428-34, Oct 1989.
- [95] M. Rao, T.-S. Jiang, and J. J.-P. Tsai. Integrated intelligent simulation environment. *Simulation*, 5(6):291-5, June 1990.
- [96] J. Rhyne. On finding conflict sets in production systems. Technical report, Department of Computer Science, University of Houston, 1977.
- [97] Y. Rim and H. Cragon. Multicache system simulation using a production system. In *Proceedings of the 1988 Southeastern Simulation Conference*, pages 64-9, October 1988.
- [98] P. Rosenbloom, J. Laird, J. McDermott, A. Newell, and E. Orciuch. R1-soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(5):561-9, 1985.
- [99] P. S. Rosenbloom. *The Chunking of Goal Hierarchies: A model of practice and stimulus-response compatibility*. PhD thesis, Computer Science Department, Carnegie Mellon University, August 1983.
- [100] D. J. Scales. Efficient matching algorithms for the SOAR/OPS5 production system. Technical Report KSL-86-47, Knowledge Systems Laboratory, Stanford University, June 1986.
- [101] J. G. Schmolze and S. Goel. A parallel asynchronous distributed production system. In *Proceedings of the National Conference on Artificial Intelligence*, pages 65-71, 1990.

- [102] F. Schreiner and G. Zimmerman. Pesa-1: A parallel architecture for production systems. In *Proceedings of the International Conference on Parallel Processing*, pages 166-9, August 1987.
- [103] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.
- [104] T. Sellis and C.-C. Lin. Performance of DBMS implementations of production systems. In *Proceedings of the Second International Conference on Tools for Artificial Intelligence*, pages 393-9, Nov 1990.
- [105] S. Stolfo. Five parallel algorithms for production system execution on the DADO machine. In *Proceedings of the National Conference on Artificial Intelligence*, pages 300-7, August 1984.
- [106] S. J. Stolfo and D. P. Miranker. The DADO production system machine. *Journal of Parallel and Distributed Computing*, 3:269-96, 1986.
- [107] S. J. Stolfo, O. Wolfson, P. K. Chan, H. M. Dewan, L. Woodbury, J. S. Glazier, and D. A. Ohsie. PARULEL: Parallel rule processing using meta-rules for redaction. *Journal of Parallel and Distributed Computing*, 13(4):366-82, December 1991.
- [108] M. Stonebraker. Integration of rule systems and database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):415-23, November 1992.
- [109] I. Subramaniam. *Managing Discardable Pages*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1993.
- [110] Sybase Corporation, Emeryville CA. *Sybase V4.0 Reference Manual*, 1990.
- [111] M. Tambe. Personal communication, 1988.
- [112] M. Tambe. *Eliminating Combinatorics from Production Match*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.
- [113] M. Tambe, D. Kalp, A. Gupta, C. Forgy, B. Milnes, and A. Newell. Soar/PSM-E: Investigating match parallelism in a learning production system. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, pages 146-60, July 1988.
- [114] M. Tambe, D. Kalp, and P. Rosenbloom. An efficient algorithm for production systems with linear-time match. In *Proceedings of the Fourth International Conference on Tools with Artificial Intelligence*, pages 36-43, November 1992.

- [115] M. Tambe and A. Newell. Some chunks are expensive. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 451-8, June 1988.
- [116] M. Tambe and P. Rosenbloom. A framework for investigating production system formulations with polynomially bounded match. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 693-700, August 1990.
- [117] M. Tambe and P. Rosenbloom. Investigating production system representations for non-combinatorial match. *Artificial Intelligence*, 68(1), August 1994.
- [118] J. Tan, M. Maheshwari, and J. Srivastava. GridMatch: A basis for integrating production systems with relational databases. In *Proceedings of the Second International Conference on Tools for Artificial Intelligence*, pages 400-7, Nov 1990.
- [119] M. F. M. Tenorio and D. E. Moldovan. Mapping production systems into multiprocessors. In *International Conference on Parallel Processing*, pages 56-62, August 1985.
- [120] D. Turner. An overview of Miranda. *SIGPLAN Notices*, December 1986.
- [121] A. Tzvieli and S. Cunningham. Super-imposing and network structure on a production system to improve performance. In *Proceedings of the IEEE workshop on Tools for artificial intelligence*, pages 345-52, Oct 1989.
- [122] M. van Biema, D. Miranker, and S. Stolfo. The Do-loop considered harmful in production system programming. In *Proceedings of the First International Conference on Expert Database Systems*, pages 177-89, 1987.
- [123] W. van Melle, A. Scott, J. Bennett, and M. Peairs. The EMYCIN manual. Technical Report STAN-CS-81-885, Department of Computer Science, Stanford University, October 1981.
- [124] D. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI-TR-271, Project MAC, Massachusetts Institute of Technology, 1972. (Reprinted in P. Winston (Ed.) 1975. *The psychology of computer vision*. McGraw-Hill, New York, 19-92).
- [125] D. Waterman and F. Hayes-Roth. *Pattern-directed Inference Systems*. Academic Press, 1978.
- [126] J. Widom, R. Cochrane, and B. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 275-85, Sept 1991.

- [127] J. Widom and S. Finkelstein. A syntax and semantics for set-oriented production rules in relational database systems. *SIGMOD Record*, 18(3):36–45, September 1989.
- [128] S.-Y. Wu and J. C. Browne. Explicit parallel structuring for rule-based programming. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 479–88, April 1993.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Carnegie Mellon University does not discriminate on the basis of race, color, sex, age, national origin, disability, or any other protected class. Information on Title VI, Title IX, Section 504, Title II of the Education Amendments of 1972 and Section 508 of the Rehabilitation Act of 1973 can be found in the state or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate on the basis of race, color, sex, age, national origin, disability, or any other protected class. Information on Title VI, Title IX, Section 504, Title II of the Education Amendments of 1972 and Section 508 of the Rehabilitation Act of 1973 can be found in the state or local laws or executive orders.

Persons concerning a question of Title VI or Title IX or Section 504 should contact the Office of Equal Opportunity, 3500 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2894, or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2894.